Data lad

Handbook

Introduction • Advanced topics • Use cases

ADINA WAGNER & MICHAEL HANKE

with

Laura Waite, Kyle Meyer, Marisa Heckner,
Benjamin Poldrack, Yaroslav Halchenko,
Chris Markiewicz, Pattarawat Chormai,
Lisa N. Mochalski, Lisa Wiersch, Jean-Baptiste Poline,
Nevena Kraljevic, Alex Waite, Lya K. Paas,
Niels Reuter, Peter Vavra, Tobias Kadelka,
Peer Herholz, Alexandre Hutton, Sarah Oliveira,
Dorian Pustina, Hamzah Hamid Baagil,
Tristan Glatard, Giulia Ippoliti, Christian Mönch,
Togaru Surya Teja, Dorien Huijser, Ariel Rokem,
Remi Gau, Judith Bomba, Konrad Hinsen,
Wu Jianxiao, Małgorzata Wierzba, Stefan Appelhoff,
Michael Joseph, Tamara Cook, Stephan Heunis,
Joerg Stadler, Sin Kim, Oscar Esteban,
Michał Szczepanik, eort, Myrskyta, Thomas Guiot

CONTENTS

Part I

Introduction

A BRIEF OVERVIEW OF DATALAD

There can be numerous reasons why you ended up with this handbook in front of you – We do not know who you are, or why you are here. You could have any background, any amount of previous experience with DataLad, any individual application to use it for, any level of maturity in your own mental concept of what DataLad is, and any motivational strength to dig into this software.

All this brief section tries to do is to provide a minimal, abstract explanation of what DataLad is, to give you, whoever you may be, some idea of what kind of tool you will learn to master in this handbook, and to combat some prejudices or presumptions about DataLad one could have.

To make it short, DataLad is a software tool developed to aid with everything related to the evolution of digital objects.

It is **not only keeping track of code**, it is **not only keeping track of data**, it is **not only making sharing, retrieving and linking data (and metadata) easy**, but it assists with the combination of all things necessary in the digital workflow of data and science.

As built-in, but *optional* features, DataLad yields FAIR resources – for example METADATA and PROVENANCE – and anything (or everything) can be easily shared *should the user want this*.

1.1 On Data

Everyone uses data. But once it exists, it does not suffice for most data to simply reside unchanged in a single location for eternity.

Most **data need to be shared** – may it be a digital collection of family photos, a genomic database between researchers around the world, or inventory lists of one company division to another. Some data are public and should be accessible to everyone. Other data should circulate only among a select few. There are various ways to distribute data, from emailing files to sending physical storage media, from pointers to data locations on shared file systems to using cloud computing or file hosting services. But what if there was an easy, **generic way of sharing and obtaining data**?

Most **data changes and evolves**. A scientist extends a data collection or performs computations on it. When applying for a new job, you update your personal CV. The documents required for an audit need to comply to a new version of a common naming standard and the data files are thus renamed. It may be easy to change data, but it can be difficult to revert a change, get information on previous states of this data, or even simply find out how a piece of data came into existence. This latter aspect, the PROVENANCE of data – information on its lineage and *how* it came to be in its current state – is often key to understanding or establishing trust in data. In collaborative fields that work with small-sized data such as Wikipedia pages or

software development, VERSION CONTROL tools are established and indispensable. These tools allow users to keep track of changes, view previous states, or restore older versions. How about a **version control system for data**?

If data are shared as a copy *of one state* of its history, **keeping all shared copies of this data up-to-date** once the original data changes or evolves is at best tedious, but likely impossible. What about ways to easily **update data and its shared copies**?

The world is full of data. The public and private sector make use of it to understand, improve, and innovate the complex world we live in. Currently, this process is far from optimal. In order for society to get the most out of public data collections, public **data need to be** FAIR[1]: Findable, Accessible, Interoperable, and Reusable. Apart from easy ways to share or update shared copies of data, extensive **metadata** is required to identify data, link data collections together, and make them findable and searchable in a standardized way. Can we also easily **attach metadata to our data and its evolution**?

DataLad is a general purpose tool for managing everything involved in the digital workflow of using data – regardless of the data's type, content, size, location, generation, or development. It provides functionality to share, search, obtain, and version control data in a distributed fashion, and it aids managing the evolution of digital objects in a way that fulfills the FAIR[2] principles.

1.2 The DataLad Philosophy

From a software point of view, DataLad is a command line tool, with an additional Python API to use its features within your software and scripts. While being a general, multi-purpose tool, there are also plenty of extensions that provide helpful, domain specific features that may very well fit your precise use case.

But beyond software facts, DataLad is built up on a handful of principles. It is this underlying philosophy that captures the spirit of what DataLad is, and here is a brief overview on it.

1. **DataLad only cares (knows) about two things: Datasets and files.** A DataLad dataset is a collection of files in folders. And a file is the smallest unit any dataset can contain. Thus, a DataLad dataset has the same structure as any directory on your computer, and DataLad itself can be conceptualized as a content-management system that operates on the units of files. As most people in any field work with files on their computer, at its core, **DataLad is a completely domain-agnostic, general-purpose tool to manage data**. You can use it whether you have a PhD in Neuroscience and want to share one of the largest whole brain MRI images in the world[3], organize your private music library, keep track of all cat memes[4] on the internet, or anything else[5].

2. **A dataset is a Git repository**. All features of the VERSION CONTROL system GIT also apply to everything managed by DataLad – plus many more. If you do not know or use Git yet, there is no need to panic – there is no necessity to learn all of Git to follow along in learning and using DataLad. You will experience much of Git working its magic underneath the hood when you use DataLad, and will soon start to appreciate its features. Later, you may want to know more on how DataLad uses Git as a fundamental layer and learn some of Git.

---

[1] https://www.go-fair.org/
[2] https://www.go-fair.org/
[3] https://github.com/datalad-datasets/bmmr-t1w-250um
[4] https://www.diabloii.net/gallery/data/500/medium/moar6-cat.jpg
[5] https://media.giphy.com/media/3o6YfXCehdioMXYbcs/giphy.gif

3. **A DataLad dataset can take care of managing and version controlling arbitrarily large data**. To do this, it has an optional *annex* for (large) file content. Thanks to this ANNEX, DataLad can easily track files that are many TB or PB in size (something that Git could not do, and allows you to transform, work with, and restore previous versions of data, while capturing all PROVENANCE, or share it with whomever you want). At the same time, DataLad does all of the magic necessary to get this awesome feature to work quietly in the background. The annex is set-up automatically, and the tool GIT-ANNEX (https://git-annex.branchable.com) manages it all underneath the hood. Worry-free large-content data management? Check!

4. Deep in the core of DataLad lies the social principle to **minimize custom procedures and data structures**. DataLad will not transform your files into something that only DataLad or a specialized tool can read. A PDF file (or any other type of file) stays a PDF file (or whatever other type of file it was) whether it is managed by DataLad or not. This guarantees that users will not lose data or access if DataLad would vanish from their system (or from the face of the Earth). Using DataLad thus does not require or generate data structures that can only be used or read with DataLad – DataLad does not tie you down, it liberates you.

5. Furthermore, DataLad is developed for **complete decentralization**. There is no required central server or service necessary to use DataLad. In this way, no central infrastructure needs to be maintained (or paid for). Your own laptop is the perfect place for your Data-Lad project to live, as is your institution's webserver, or any other common computational infrastructure you might be using.

6. Simultaneously, though, DataLad aims to **maximize the (re-)use of existing 3rd-party data resources and infrastructure**. Users *can* use existing central infrastructures should they want to. DataLad works with any infrastructure from GITHUB to Dropbox[6], Figshare[7] or institutional repositories, enabling users to harvest all of the advantages of their preferred infrastructure without tying anyone down to central services.

These principles hopefully gave you some idea of what to expect from DataLad, cleared some worries that you might have had, and highlighted what DataLad is and what it is not. The section *What you really need to know* (page 27) will give you a one-page summary of the functionality and commands you will learn with this handbook. But before we get there, let's get ready to *use* DataLad. For this, the next section will show you how to use the handbook.

---

[6] https://www.dropbox.com
[7] https://figshare.com/

HOW TO USE THE HANDBOOK

2.1 For whom this book is written

The DataLad handbook is not the DataLad documentation, and it is also not an explanation of the computational magic that happens in the background. Instead, it is a procedurally oriented, hands-on crash-course that invites you to fire up your terminal and follow along.

If you are interested in learning how to use DataLad, this handbook is for you.

You do not need to be a programmer, computer scientist, or Linux-crank. If you have never touched your computer's shell before, you will be fine. No knowledge about GIT or GIT-ANNEX is required or necessary. Regardless of your background and personal use cases for DataLad, the handbook will show you the principles of DataLad, and from chapter 1 onwards you will be using them.

2.2 How to read this book

First of all: be excited. DataLad can help you to manage your digital data workflow in various ways, and in this book you will use many of them right from the start. There are many topics you can explore, if you wish: local or collaborative workflows, reproducible analyses, data publishing, and so on. If anything seems particularly exciting, you can go ahead, read it, *and do it*. Therefore, **grab your computer, and be ready to use it**.

Every chapter will give you different challenges, starting from basic local workflows to more advanced commands, and you will see your skills increase with each. While learning, it will be easy to **find use cases in your own work for the commands you come across**.

Throughout the book numerous *terms* for concepts and technical components are used. They are all defined in a *Glossary* (page 471), and are printed in small-caps, such as GIT, or COMMIT MESSAGE.

As the handbook is to be a practical guide it includes as many hands-on examples as we can fit into it. Code snippets look like this, and you should **copy them into your own terminal to try them out**, but you can also **modify them to fit your custom needs in your own use cases**. Note how we distinguish `comments (#)` from `commands ($)` and their output in the example below (it shows the creation of a DataLad dataset):

```
# this is a comment used for additional explanations.
# Anything preceded by $ is a command to try.
# if the line starts with neither # nor $, its the output of a command
$ datalad create myfirstrepo
```

```
[INFO   ] Creating a new annex repo at /home/adina/DataLad-101
create(ok): /home/adina/DataLad-101 (dataset)
```

When copying code snippets into your own terminal, do not copy the leading $ – this only indicates that the line is a command, and would lead to an error when executed. Don't worry *if you do not want to code along* (page 9), though.

Instead of copying manually, you can also click on the clipboard icon at the top right of each code snippet. Clicking on that icon will copy all relevant lines from the code snippet, and will drop all comments and the $ automatically.

The book is split into different parts. The upcoming chapters are the *Basics* that intend to show you the core DataLad functionality and challenge you to use it. If you want to learn how to use DataLad, it is recommended to start with this part and read it from start to end. In the part *use cases*, you will find concrete examples of DataLad applications for general inspiration – this is the second part of this book. If you want to get an overview of what is possible with DataLad, this section will show you in a concise and non-technical manner. Pick whatever you find interesting and disregard the rest. Afterwards, you might even consider *Contributing* (page 498) to this book by sharing your own use case.

Note that many challenges can have straightforward and basic solutions, but a lot of additional options or improvements are possible. Sometimes one could get lost in all of the available DataLad functionality, or in some interesting backgrounds about a command. For this reason we put all of the basics in plain sight, and those basics will let you master a given task and get along comfortably. Having the basics will be your multi-purpose swiss army knife. But if you want to have the special knowledge for a very peculiar type of problem set or that extra increase in skill or understanding, you'll have to do a detour into some of the "hidden" parts of the book: When there are command options or explanations that go beyond basics and best practices, we put them in special boxes in order to not be too distracting for anyone only interested in the basics. You can decide for yourself whether you want to check them out:

"Find-out-more" boxes contain general additional information:

> **M2.1 For curious minds**
>
> Sections like this contain content that goes beyond the basics necessary to complete a challenge.

"Git user notes" elaborate on technical details from under the hood:

> **G2.1 For (future) Git experts**
>
> DataLad uses GIT and GIT-ANNEX underneath the hood. Readers that are familiar with these tools can find occasional notes on how a DataLad command links to a Git(-annex) command or concept in boxes like this. There is, however, absolutely no knowledge of Git or git-annex necessary to follow this book. You will, though, encounter Git commands throughout the book when there is no better alternative, and executing those commands will suffice to follow along.

If you are a Windows 10 user with a native (i.e., not Windows Subsystem for Linux (WSL)[8]-based DataLad installation, pay close attention to the special notes in so-called "Windows-Wits":

---

[8] https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux

> **W2.1 For Windows users only**
>
> A range of file system issues can affect the behavior of DataLad or its underlying tools on Windows 10. If necessary, the handbook provides workarounds for problems, explanations, or at least apologies for those inconveniences. If you want to help us make the handbook or DataLad better for Windows users, please get in touch[9] – every little improvement or bug report can help.
>
> ───────────────
> [9] https://github.com/datalad-handbook/book/issues/new/

Apart from core DataLad commands (introduced in the *Basics* part of this book), DataLad also comes with many extensions and advanced commands not (yet) referenced in this handbook. The development of many of these features is ongoing, and this handbook will incorporate all DataLad commands and extensions *once they are stable* (that is, once the command(-structure) is likely not to change anymore). If you are looking for a feature but cannot find it in this handbook, please take a look at the documentation[10], write[11] or request[12] an additional chapter if you believe it is a worthwhile addition, or ask a question on Neurostars.org[13] with a `datalad` tag if you need help.

What you will learn in this book

This handbook will teach you simple, yet advanced principles of data management for reproducible, comprehensible, transparent, and FAIR[14] data projects. It does so with hands-on tool use of DataLad and its underlying software, blended with clear explanations of relevant theoretical backgrounds whenever necessary, and by demonstrating organizational and procedural guidelines and standards for data related projects on concrete examples.

You will learn how to create, consume, structure, share, publish, and use *DataLad datasets*: modular, reusable components that can be version-controlled, linked, and that are able to capture and track full provenance of their contents, if used correctly.

At the end of the `Basics` section, these are some of the main things you will know how to do, and understand why doing them is useful:

- **Version-control** data objects, regardless of size, keep track of and **update** (from) their sources and shared copies, and capture the **provenance** of all data objects whether you consume them from any source or create them yourself.

- **Build up complete projects** with data as independent, version-controlled, provenance-tracked, and linked DataLad dataset(s) that allow **distribution**, modular **reuse**, and are **transparent** both in their structure and their development to their current and future states.

- **Bind** modular components into complete data analysis projects, and comply to procedural and organizational principles that will help to create transparent and comprehensible projects to ease **collaboration** and **reproducibility**.

- **Share** complete data objects, version-controlled as a whole, but including modular components (such as data) in a way that preserves the history, provenance, and linkage of its

───────────────
[10] http://docs.datalad.org
[11] http://handbook.datalad.org/en/latest/contributing.html
[12] https://github.com/datalad-handbook/book/issues/new
[13] https://neurostars.org/latest
[14] https://www.go-fair.org/fair-principles/

components.

After having read this handbook, you will find it easy to create, build up, and share intuitively structured and version-controlled data projects that fulfill high standards for reproducibility and FAIRness. You are able to decide for yourself how deep you want to delve into the DataLad world based on your individual use cases, and with every section you will learn more about state-of-the-art data management.

The storyline

Most of the sections in the upcoming chapter follow a continuous **narrative**. This narrative aims to be as domain-agnostic and relatable as possible, but it also needs to be able to showcase all of the principles and commands of DataLad. Therefore, together we will build up a DataLad project for the fictional educational course `DataLad-101`.

Envision yourself in the last educational course you took or taught. You have probably created some files with notes you took, a directory with slides or books for further reading, and a place where you stored assignments and their solutions. This is what we will be doing as well. This project will start with creating the necessary directory structures, populating them by `installing` and `creating` several DATALAD SUBDATASETs, adding files and changing their content, and executing simple scripts with input data to create results we can share and publish with DataLad.

2.3 Let's get going!

If you have DataLad installed, you can dive straight into chapter 1, *Create a dataset* (page 34). For everyone new, there are the sections *General prerequisites* (page 22) as a minimal tutorial to using the shell and *Installation and configuration* (page 10) to get your DataLad installation set up.

> **M2.2 I can not/do not want to code along...**
>
> If you do not want to follow along and only read, there is a showroom dataset of the complete DataLad-101 project at github.com/datalad-handbook/DataLad-101[15]. This dataset contains a separate BRANCH for each section that introduced changes in the repository. The branches have the names of the sections, e.g., sct_create_a_dataset marks the repository state at the end of the first section in the first chapter. You can checkout a branch with *git checkout <branch-name>* to explore how the dataset looks like at the end of a given section.
>
> Note that this "public" dataset has a number of limitations, but it is useful for an overview of the dataset history (and thus the actions performed throughout the "course"), a good display of how many and what files will be present in the end of the book, and a demonstration of how subdatasets are linked.
>
> ---
> [15] https://github.com/datalad-handbook/DataLad-101

INSTALLATION AND CONFIGURATION

3.1 Install DataLad

> ⚠️ **Feedback on installation instructions**
>
> The installation methods presented in this chapter are based on experience and have been tested carefully. However, operating systems and other software are continuously evolving, and these guides might have become outdated. Please file an issue[16], if you encounter problems installing DataLad, and help keeping this information up-to-date.
>
> ---
> [16] https://github.com/datalad-handbook/book/issues/new

In general, the DataLad installation requires Python 3 (see the *Find-out-more* 💡 *M3.1 on the difference between Python 2 and 3* (page 11) to learn why this is required), GIT, and GIT-ANNEX, and for some functionality 7-Zip[17]. The instructions below detail how to install the core DataLad tool and its dependencies on common operating systems. They do not cover the various *DataLad extensions* (page 285) that need to be installed separately, if desired.

The following sections provide targeted installation instructions for a set of common scenarios, operating systems, or platforms.

Windows 10

There are countless ways to install software on Windows. Here we describe *one* possible approach that should work on any Windows computer, like one that you may have just bought.

Python: Windows itself does not ship with Python, it must be installed separately. If you already did that, please check the *Find-out-more* 💡 *M3.1 on Python versions* (page 11), if it

---
[17] https://7-zip.org/

> **M3.1 Python 2, Python 3, what's the difference?**
>
> DataLad requires Python 3.6, or a more recent version, to be installed on your system. The easiest way to verify that this is the case is to open a terminal and type **python** to start a Python session:
>
> ```
> $ python
> Python 3.9.1+ (default, Jan 20 2021, 14:49:22)
> [GCC 10.2.1 20210110] on linux
> Type "help", "copyright", "credits" or "license" for more information.
> >>>
> ```
>
> If this fails, or reports a Python version with a leading 2, such as Python 2.7.18, try starting **python3**, which some systems use to disambiguate between Python 2 and Python 3. If this fails, too, you need to obtain a recent release of Python 3. On Windows, attempting to run commands that are not installed might cause a Windows Store window to pop up. If this happens, it means you have no Python installed. Please check the *Windows 10* (page 10) installation instructions, and *do not* install Python via the Windows Store.
>
> Python 2 is an outdated, in technical terms "deprecated", version of Python. Although it still exist as the default Python version on many systems, it is no longer maintained since 2020, and thus, most software has dropped support for Python 2. If you only run Python 2 on your system, most Python software, including DataLad, will be incompatible, and hence unusable, resulting in errors during installation and execution.
>
> But does that mean that you should uninstall Python 2? **No!** Keep it installed, especially if you are using Linux or MacOS. Python 2 existed for 20 years and numerous software has been written for it. It is quite likely that some basic operating system components or legacy software on your computer is depending on it, and uninstalling a preinstalled Python 2 from your system will likely render it unusable. Install Python 3, and have both versions coexist peacefully.

matches the requirements. Otherwise, head over to the download section of the Python website[18], and download an installer. Unless you have specific requirements, go with the 64bit installer of the latest Python 3 release.

> **W3.1 Avoid installing Python from the Windows store**
>
> We recommend to **not** install Python via the Windows store, even if it opens after you typed `python`, as this version requires additional configurations by hand (in particular of your $PATH ENVIRONMENT VARIABLE).

When you run the installer, make sure to select the *Add Python to PATH* option, as this is required for subsequent installation steps and interactive use later on. Other than that, using the default installation settings is just fine.

Git: Windows also does not come with Git. If you happen to have it installed already, please check, if you have configured it for command line use. You should be able to open the Windows command prompt and run a command like `git --version`. It should return a version number and not an error.

To install Git, visit the Git website[19] and download an installer. If in doubt, go with the 64bit installer of the latest version. The installer itself provides various customization options. We recommend to leave the defaults as they are, in particular the target directory, but configure the following settings (they are distributed over multiple dialogs):

- Enable *Use a TrueType font in all console windows*
- Select *Git from the command line and also from 3rd-party software*
- *Enable file system caching*
- *Enable symbolic links*

Git-annex: The most convenient way to deploy git-annex is via the DataLad installer[20]. Once Python is available, it can be installed with the Python package manager `pip`. Open a command prompt and run:

```
> pip install datalad-installer
```

Afterwards, open another command prompt in administrator mode and run:

```
> datalad-installer git-annex -m datalad/packages
```

This will download a recent git-annex, and configure it for your Git installation. The admin command prompt can be closed afterwards, all other steps do not need it.

For performance improvements, we recommend to also set the following git-annex configuration:

```
> git config --global filter.annex.process "git-annex filter-process"
```

DataLad: With Python, Git, and git-annex installed, DataLad can be installed, and later also upgraded using `pip` by running:

---

[18] https://www.python.org/downloads
[19] https://git-scm.com/download/win
[20] https://github.com/datalad/datalad-installer

```
> pip install datalad
```

7-Zip (optional, but highly recommended): Download it from the 7-zip website[21] (64bit installer when in doubt), and install it into the default target directory.

There are many other ways to install DataLad on Windows, check for example the *Windows-wit* ⊞ *W3.2 on the Windows Subsystem 2 for Linux* (page 13). One particularly attractive approach is *Conda* (page 19). However, at the moment git-annex is not available from Conda on Windows. If you want to use Conda, perform the *Conda* (page 19)-based DataLad installation first, and then install git-annex via the DataLad installer, as described above.

> ⊞ **W3.2 Install DataLad using the Windows Subsystem 2 for Linux**
>
> With the Windows Subsystem for Linux, you will be able to use a Unix system despite being on Windows. You need to have a recent build of Windows 10 in order to get WSL2 – we do not recommend WSL1.
>
> You can find out how to install the Windows Subsystem for Linux at docs.microsoft.com[22]. Afterwards, proceed with your installation as described in the installation instructions for Linux.
>
> ---
> [22] https://docs.microsoft.com/en-us/windows/wsl/install-win10

Using DataLad on Windows has a few peculiarities. There is a dedicated summary, *So... Windows... eh?* (page 490) with an overview. In general, DataLad can feel a bit sluggish on non-WSL2 Windows systems. This is due to various filesystem issues that also affect the version control system GIT itself, which DataLad relies on. The core functionality of DataLad works, and you should be able to follow most contents covered in this book. You will notice, however, that some Unix commands displayed in examples may not work, and that terminal output can look different from what is displayed in the code examples of the book, and that some dependencies for additional functionality are not available for Windows. If you are a Windows user and want to help improve the handbook for Windows users, please get in touch[23]. Dedicated notes, "Windows-wits", contain important information, alternative commands, or warnings. If you on a native Windows 10 system, you should pay close attention to them.

Mac OSX

Modern Macs come with a compatible Python 3 version installed by default. The *Find-out-more* 💡 *M3.1 on Python versions* (page 11) has instructions on how to confirm that.

DataLad is available via OS X's homebrew[24] package manager. First, install the homebrew package manager, which requires Xcode[25] to be installed from the Mac App Store.

Next, install datalad and its dependencies:

```
$ brew install datalad
```

Likewise, the optional, but recommended, p7zip[26] dependency can be installed via **brew** as well.

---

[21] https://7-zip.org
[23] https://github.com/datalad-handbook/book/issues/new
[24] https://brew.sh
[25] https://apps.apple.com/us/app/xcode/id497799835
[26] http://p7zip.sourceforge.net/

Alternatively, you can exclusively use **brew** for DataLad's non-Python dependencies, and then check the *Find-out-more* 💡 *M3.2 on how to install DataLad via Python's package manager* (page 15).

3.2 Installation instructions for the JSC (JURECA and JUDAC)

Read this if you want to use DataLad at the JSC.

Software installation

One common INM-7 use case for DataLad is using DataLad in conjunction with the datalad-containers extension on JUDAC[27] and JURECA[28]. Due to inode limitations[29], the installation of all relevant software needs to create as few files as possible. Else, a default installation on these two systems will exhaust a user's inode limit completely, preventing the creation of any additional files. In order to avoid this, the installation needs to proceed in a way that is shared between JURECA and JUDAC:

1. Once you have a JuDoor[30] account, log into JUDAC:

   ```
   $ ssh <user-ID>@judac.fz-juelich.de
   ```

2. Download the latest Miniconda installer and install it into the shared/ directory. While you have separate $HOME directories on both HPC systems, shared/ is a directory that both systems can access:

   ```
   $ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O␣
   ↪shared/Miniconda3-latest-Linux-x86_64.sh
   $ bash shared/Miniconda3-latest-Linux-x86_64.sh -p ~/shared/miniconda3
   # An interactive installer will ask you to read and agree to the
   # license terms, and will ask you to confirm the installation location.
   # reply "yes" when asked whether to perform a conda init
   ```

3. Open a new shell. Install all required software via conda:

   ```
   $ conda install -c conda-forge datalad datalad-container p7zip
   ```

Subsequently, DataLad and all relevant software should be available on JUDAC, and you should be able to run any DataLad command from the terminal. This is because the final part of the Miniconda installation should have adjusted your .bashrc file such that conda and all software installed via conda install is accessible from the command line.

💡 **M3.3 What's this modification exactly?**

It looks like this:

---

[27] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/JUDAC/JUDAC_node.html

[28] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html

[29] https://www.fz-juelich.de/SharedDocs/FAQs/IAS/JSC/EN/JUST/FAQ_01_Data_limitiations.html?nn=1765188

[30] https://judoor.fz-juelich.de/login

> ### M3.2 Install DataLad via pip on MacOSX
>
> If Git/git-annex are installed already (via brew), DataLad can also be installed via Python's package manager `pip`, which should be installed by default on your system:
>
> ```
> $ pip install datalad
> ```
>
> Recent macOS versions may use `pip3` instead of `pip` – use TAB COMPLETION to find out which is installed.
> Recent macOS versions may warn after installation that scripts were installed into locations that were not on PATH:
>
> ```
> The script chardetect is installed in
> '/Users/MYUSERNAME/Library/Python/3.7/bin' which is not on PATH.
> Consider adding this directory to PATH or, if you prefer to
> suppress this warning, use --no-warn-script-location.
> ```
>
> To fix this, add these paths to the `$PATH` environment variable. You can either do this for your own user (1), or for all users of the computer (2) (requires using sudo and authenticating with your computer's password):
>
> (1) Add something like (exchange the user name accordingly)
>
> ```
> export PATH=$PATH:/Users/MYUSERNAME/Library/Python/3.7/bin
> ```
>
> to the *profile* file of your shell. If you use a BASH shell, this may be `~/.bashrc` or `~/.bash_profile`, if you are using a ZSH shell, it may be `~/.zshrc` or `~/.zprofile`. Find out which shell you are using by typing `echo $SHELL` into your terminal.
>
> (2) Alternatively, configure it *system-wide*, i.e., for all users of your computer by adding the the path `/Users/MYUSERNAME/Library/Python/3.7/bin` to the file `/etc/paths`, e.g., with the editor NANO:
>
> ```
> sudo nano /etc/paths
> ```
>
> The contents of this file could look like this afterwards (the last line was added):
>
> ```
> /usr/local/bin
> /usr/bin
> /bin
> /usr/sbin
> /sbin
> /Users/MYUSERNAME/Library/Python/3.7/bin
> ```

```
$ cat .bashrc
[...]
#
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/p/home/jusers/<user-ID>/judac/shared/miniconda3/bin/conda'
↪'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/p/home/jusers/<user-ID>/judac/shared/miniconda3/etc/profile.d/conda.sh
↪" ]; then
        . "/p/home/jusers/<user-ID>/judac/shared/miniconda3/etc/profile.d/conda.sh"
    else
        export PATH="/p/home/jusers/<user-ID>/judac/shared/miniconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

It may look a bit messy if you are unfamiliar with bash, but worry not – it simply points your shell to the location of all conda-installed programs so that their commands become available to you.

To get everything to work on JURECA as well requires that your `.bashrc` file on JURECA gets the same modifications. This is some extra work, but done in a few command line calls:

1. Adjust paths in the `.bashrc` file to remove `judac/` with the stream editor sed[31]:

   ```
   $ sed -i 's/judac\/shared\/miniconda3\//shared\/miniconda3\//' .bashrc
   ```

2. Move the `.bashrc` file into `shared/`, and create a SYMLINK in its place:

   ```
   $ mv .bashrc shared/
   # create a symlink
   $ ln -s shared/.bashrc .bashrc
   ```

3. Log out of JUDAC and log into JURECA from your local machine:

   ```
   $ ssh <user-ID>@jureca.fz-juelich.de
   ```

4. Make sure that your `.bashrc` on JURECA does not contain any precious content. It should look something like this:

   ```
   $ cat .bashrc
   # *****************************************************************************
   # bash environment file in $HOME
   # Please see:
   # http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/OnlineStorage/JUST/
   ↪FAQ/just-FAQ_node.html
   # for more information and possible modifications to this file
   # *****************************************************************************

   # Source global definitions: Copied from CentOS 7 /etc/skel/.bashrc
   ```

   (continues on next page)

---

[31] https://www.gnu.org/software/sed/manual/sed.html

```
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
```

> 💡 **M3.4 What if it contains other content than this?**
>
> The content in the .bashrc file above is not precious, because the .bashrc file you placed into shared should already contain them. If there is more, e.g., configurations you made yourself, make sure that you copy and paste them into the .bashrc file in shared/. Usually, there should be no need to adjust paths.

5. Remove the .bashrc file and symlink the .bashrc file in shared/ instead:

```
$ rm .bashrc
$ ln -s shared/.bashrc .bashrc
```

6. Open a new session on JURECA. You should now have access to the software you just installed on JUDAC.

> 💡 **M3.5 Troubleshooting inode quotas**
>
> The inode limit from the JSC is quite strict. If you receive an e-mail that you have exceeded your quota, here is what you can do:
> - Verify that it is inode limitations that you ran into by running jutil user dataquota -u <user-ID>. Check the table columns "inode-usage" and "inode-<soft|hard>-limit".
> - Check that your installation does not consume more inodes than expected. On JURECA, cd into the shared/ directory and run the ncdu command. Once the command finished scanning, press c and confirm that your miniconda3 directory consumes about 40k inodes.
> - Remove caches and unused packages by running conda clean --all to reduce the inode usage by a few thousand.
> - On JURECA, run ncdu in your $HOME directory to check whether there are other directories that consume many inodes.
>
> The installation takes up almost all available inodes, so be aware that you can only have a few thousand files in any of the two systems $HOME directories.

Configurations on JURECA und JUDAC

In order to use DataLad, it is highly recommended to configure your Git identity. While it is not strictly *necessary*, it makes sense to do it in a way that is shared between the two HPC systems as well.

On any of the two systems, provide your Name and e-mail address to the **git config** command:

```
$ git config --global --add user.name "Bob McBobFace"
$ git config --global --add user.email bob@example.com
```

This will create a .gitconfig file in your $HOME directory. Just as done with the .bashrc file, move this file into the shared/ directory, and create a symlink in its place:

```
$ mv .gitconfig shared/
$ ln -s shared/.gitconfig .gitconfig
$ logout
# log into the other machine
$ ssh <user-ID>@<jureca|judac>.fz-juelich.de
# create a symlink to the shared .gitconfig file
$ ln -s shared/.gitconfig .gitconfig
```

Afterwards, you are done, and ready to use DataLad on the HPC systems of the JSC.

3.3 Standard installation instructions

Read this, if you want to install DataLad on your own computer, or any system that is not part of the JSC.

Linux: (Neuro)Debian, Ubuntu, and similar systems

DataLad is part of the Debian and Ubuntu operating systems. However, the particular DataLad version included in a release may be a bit older (check the versions for Debian[32] and Ubuntu[33] to see which ones are available).

For some recent releases of Debian-based operating systems, NeuroDebian[34] provides more recent DataLad versions (check the availability table[35]). In order to install from NeuroDebian, follow its installation documentation[36], which only requires copy-pasting three lines into a terminal. Also, should you be confused by the name: enabling this repository will not do any harm if your field is not neuroscience.

Whichever repository you end up using, the following command installs DataLad and all of its software dependencies (including GIT-ANNEX and p7zip[37]):

```
$ sudo apt-get install datalad
```

The command above will also upgrade existing installations to the most recent available version.

Linux: CentOS, Redhat, Fedora, or similar systems

For CentOS, Redhat, Fedora, or similar distributions, there is an RPM package for git-annex[38]. A suitable version of Python and GIT should come with the operating system, although some servers may run fairly old releases.

DataLad itself can be installed via `pip`:

```
$ pip install datalad
```

---

[32] https://packages.debian.org/datalad
[33] https://packages.ubuntu.com/datalad
[34] http://neuro.debian.net
[35] http://neuro.debian.net/pkgs/datalad.html
[36] http://neuro.debian.net/install_pkg.html?p=datalad
[37] http://p7zip.sourceforge.net/
[38] https://git-annex.branchable.com/install/rpm_standalone/

Alternatively, DataLad can be installed together with GIT and GIT-ANNEX via *Conda* (page 19) as outlined in the section below.

Linux-machines with no root access (e.g. HPC systems)

The most convenient user-based installation can be achieved via *Conda* (page 19).

Conda

Conda is a software distribution available for all major operating systems, and its Miniconda[39] installer offers a convenient way to bootstrap a DataLad installation. Importantly, it does not require admin/root access to a system.

Detailed, platform-specific installation instruction[40] are available in the Conda documentation. In short: download and run the installer, or, from the command line, run

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-<YOUR-OS>-x86_64.sh
$ bash Miniconda3-latest-<YOUR-OS>-x86_64.sh
```

In the above call, replace `<YOUR-OS>` with an identifier for your operating system, such as "Linux" or "MacOSX". During the installation, you will need to accept a license agreement (press Enter to scroll down, and type "yes" and Enter to accept), confirm the installation into the default directory, and you should respond "yes" to the prompt "`Do you wish the installer to initialize Miniconda3 by running conda init? [yes|no]`". Afterwards, you can remove the installation script by running `rm ./Miniconda3-latest-*-x86_64.sh`.

The installer automatically configures the shell to make conda-installed tools accessible, so no further configuration is necessary. Once Conda is installed, the DataLad package can be installed from the `conda-forge` channel:

```
$ conda install -c conda-forge datalad
```

In general, all of DataLad's software dependencies are automatically installed, too. This makes a conda-based deployment very convenient. A from-scratch DataLad installation on a HPC system, as a normal user, is done in three lines:

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash Miniconda3-latest-Linux-x86_64.sh
# acknowledge license, keep everything at default
$ conda install -c conda-forge datalad
```

In case a dependency is not available from Conda (e.g., there is no git-annex package for Windows in Conda), please refer to the platform-specific instructions above.

To update an existing installation with conda, use:

```
$ conda update datalad
```

---

[39] https://docs.conda.io/en/latest/miniconda.html
[40] https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html

> **W3.3 Install Unix command-line tools on Windows with Conda**
>
> On Windows, many Unix command-line tools such as `cp` that a frequently used in this handbook are not available by default. You can get a good set of tools by installing CONDAs m2-base package via `conda install m2-base`.

The DataLad installer[41] also supports setting up a Conda environment, in case a suitable Python version is already available.

Using Python's package manager `pip`

As mentioned above, DataLad can be installed via Python's package manager pip[42]. `pip` comes with any Python distribution from python.org[43], and is available as a system-package in nearly all GNU/Linux distributions.

If you have Python and `pip` set up, to automatically install DataLad and most of its software dependencies, type

```
$ pip install datalad
```

If this results in a `permission denied` error, you can install DataLad into a user's home directory:

```
$ pip install --user datalad
```

On some systems, in particular macOS, you may need to call `pip3` instead of `pip`:

```
$ pip3 install datalad
# or, in case of a "permission denied error":
$ pip3 install --user datalad
```

An existing installation can be upgraded with `pip install -U datalad`.

`pip` is not able to install non-Python software, such as 7-zip or GIT-ANNEX. But you can install the DataLad installer[44] via a `pip install datalad-installer`. This is a command-line tool that aids installation of DataLad and its key software dependencies on a range of platforms.

3.4 Initial configuration

Initial configurations only concern the setup of a GIT identity. If you are a Git-user, you should hence be good to go.

If you have not used the version control system Git before, you will need to tell Git some information about you. This needs to be done only once. In the following example, exchange `Bob McBobFace` with your own name, and `bob@example.com` with your own email address.

```
# enter your home directory using the ~ shortcut
% cd ~
% git config --global --add user.name "Bob McBobFace"
% git config --global --add user.email bob@example.com
```

---

[41] https://github.com/datalad/datalad-installer
[42] https://pip.pypa.io/en/stable/
[43] https://www.python.org
[44] https://github.com/datalad/datalad-installer

This information is used to track changes in the DataLad projects you will be working on. Based on this information, changes you make are associated with your name and email address, and you should use a real email address and name – it does not establish a lot of trust nor is it helpful after a few years if your history, especially in a collaborative project, shows that changes were made by Anonymous with the email youdontgetmy@email.fu. And do not worry, you won't get any emails from Git or DataLad.

GENERAL PREREQUISITES

DataLad uses command-line arguments in a *terminal*. This means that there is no graphical user interface with buttons to click on, but a set of commands and options users type into their shell. If you are not used to working with command-line arguments, DataLad can appear intimidating. Luckily, the set of possible commands is limited, and even without prior experience with a shell, one can get used to it fairly quickly.

This chapter aims at providing novices with general basics about the shell, common Unix commands, and some general file system facts. This chapter is also a place to return to and (re-)read if you come across a non-DataLad command or principle you want to remind yourself of. If you are already familiar with the shell and know the difference between an absolute and a relative path, you can safely skip this chapter and continue to the DataLad Basics.

Almost all of this chapter is based on parts of a wonderful lab documentation Alex Waite wrote.

4.1 The Command Line

The shell (sometimes also called a terminal, console, or CLI) is an interactive, text based interface. If you have used Matlab or IPython, then you are already familiar with the basics of a command line interface.
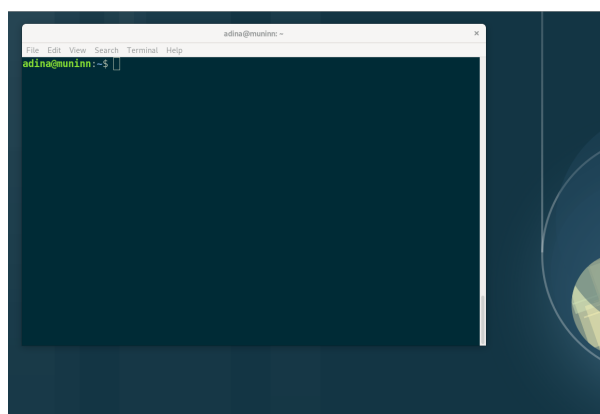
Fig. 4.1: A terminal window in a standard desktop environment.

4.2 Command Syntax

Commands are case sensitive and follow the syntax of: command [options...] <arguments...>. Whenever you see some example code in the code snippets of this book, make sure that you capitalize exactly as shown if you try it out yourself. The options modify the behavior of the program, and are usually preceded by - or --. In this example

```
$ ls -l output.txt
-rw-r--r-- 1 adina adina 25165824 Jul 29 10:07 output.txt
```

ls is the *command*. The *option* -l tells ls to use a long listing format and thus display more information. output.txt is the *argument* — the file that ls is listing. The difference between options preceded by - and -- is their length: Usually, all options starting with a single dash are single letters. Often, a long, double-dashed option exists for these short options as well. For example, to list the size of a file in a *human-readable* format, supply the short option -h, or, alternatively, its longer form, --human-readable.

```
$ ls -lh output.txt    # note that short options can be combined!
# or alternatively
$ ls -l --human-readable output.txt
-rw-r--r-- 1 adina adina 24M Jul 29 10:07 output.txt
```

Every command has many of those options (often called "flags") that modify their behavior. There are too many to even consider memorizing. Remember the ones you use often, and the rest you will lookup in their documentation or via your favorite search engine. DataLad commands naturally also come with many options, and in the next chapters and later examples you will get to see many of them.

4.3 Basic Commands

The following commands can appear in our examples or are generally useful to know: They can help you to *explore and navigate* in your file system (cd, ls), copy, move, or remove files (cp, mv, rm), or create new directories (mkdir).

ls -lah <folder> list the contents of a folder, including hidden files (-a), and all their information (-l); print file sizes in human readable units (-h)

cd <folder> change to another folder

cp <from> <to> copy a file

cp -R <from> <to> copy a folder and its contents (-R)

mv <from> <to> move/rename a file or folder

rm <file> delete a file

rm -Rv <folder> delete a folder and its contents (-R) and list each file as it's being deleted (-v)

mkdir <folder> create a folder

rmdir <folder> delete an empty folder

4.4 The Prompt

When you first login on the command line, you are greeted with "the prompt", and it will likely look similar to this:

```
adina@muninn: ~$
```

This says I am the user adina on the machine muninn and I am in the folder ~, which is shorthand for the current user's home folder (in this case /home/adina).

The $ sign indicates that the prompt is interactive and awaiting user input. In this handbook, we will use $ as a shorthand for the prompt, to allow the reader to quickly differentiate between lines containing commands vs the output of those commands.

4.5 Paths

Let's say I want to create a new folder in my home folder, I can run the following command:

```
$ mkdir /home/adina/awesome_datalad_project
```

And that works. /home/adina/awesome_datalad_project is what is called an *absolute* path. Absolute paths *always* start with a /, and define the folder's location with no ambiguity.

However, much like in spoken language, using someone's full proper name every time would be exhausting, and thus pronouns are used.

This shorthand is called *relative* paths, because they are defined (wait for it...) *relative* to your current location on the file system. Relative paths *never* start with a /.

Unix knows a few shortcuts to refer to file system related directories, and you will come across them often. Whenever you see a ., .., or ~ in a DataLad command, here is the translation to this cryptic punctuation:

. the current directory

.. the parent directory

~ the current user's home directory

So, taking the above example again: given that I am in my home (~) folder, the following commands all would create the new folder in the exact same place.

```
mkdir /home/adina/awesome_datalad_project
mkdir ~/awesome_datalad_project
mkdir awesome_datalad_project
mkdir ./awesome_datalad_project
```

To demonstrate this further, consider the following: In my home directory /home/adina I have added a folder for my current project, awesome_datalad_project/. Let's take a look at how this folder is organized:

```
$ tree
└── home
    └── adina
        └── awesome_datalad_project
            ├── aligned
```

```
    ├── code
    └── sub-01
        └── bold3T
    ├── ...
    └── sub-xx
        └── bold3T
```

Now let's say I want to change from my home directory `/home/adina` into the `code/` folder of the project. I could use absolute paths:

cd /home/adina/awesome_datalad_project/aligned/code

But that is a bit wordy. It is much easier with a relative path:

$ cd awesome_datalad_project/aligned/code

Relative to my starting location (`/home/adina`), I navigated into the subfolders.

I can change back to my home directory also with a relative path:

$ cd ../../../

The first `../` takes me from `code/` to its parent `aligned/`, the second `../` to `awesome_datalad_project/`, and the last `../` back to my home directory `adina/`.

However, since I want to go back to my home folder, it's much faster to run:

$ cd ~

4.6 Text Editors

Text editors are a crucial tool for any Linux user, but regardless of your operating system, if you use DataLad, you will occasionally find yourself in your default text editor to write a COMMIT MESSAGE to describe a change you performed in your DataLad dataset.

Religious wars have been fought over which is "the best" editor. From the smoldering ashes, this is the breakdown:

nano Easy to use; medium features. If you do not know which to use, start with this.

vim Powerful and light; lots of features and many plugins; steep learning curve. Two resources to help get the most out of vim are the vimtutor program and vimcasts.org. If you accidentally enter `vim` unprepared, typing `:q` will get you out of there.

emacs Powerful; tons of features; written in Lisp; huge ecosystem; advanced learning curve.

4.7 Shells

Whenever you use the command line on a Unix-based system, you do that in a command-line interpreter that is referred to as a `shell`.

The shell is used to start commands and display the output of those commands. It also comes with its own primitive (yet surprisingly powerful) scripting language.

Many shells exist, though most belong to a family of shells called "Bourne Shells" that descend from the original `sh`. This is relevant, because they share (mostly) a common syntax.

Two common shells are:

Bash The bourne-again shell (`bash`) is the default shell on many *nix systems (most Linux distros, MacOS).

zsh The Z shell (`zsh`) comes with many additional features, the highlights being: shared history across running shells, smarter tab-completion, spelling correction, and better theming.

To determine what shell you're in, run the following:

```
$ echo $SHELL
usr/bin/bash
```

4.8 Tab Completion

One of the best features ever invented is tab completion. Imagine your favorite animal sitting on your shoulder. Now imagine that animal shouting "TAB!" every time you've typed the first 3 letters of a word. Listen to that animal.

Tab completion autocompletes commands and paths when you press the Tab key. If there are multiple matching options, pressing Tab twice will list them.

The greatest advantage of tab completion is not increased speed (though that is a nice benefit) but rather the near elimination of typos — and the resulting reduction of cognitive load. You can actually focus on the task you're working on, rather than your typing. Tab-completion will autocomplete a DataLad command, options you give to it, or paths.

For an example of tab-completion with paths, consider the following directory structure:

```
├── Desktop
├── Documents
│   ├── my_awesome_project
│   └── my_comics
│       └── xkcd
│           └── is_it_worth_the_time.png
├── Downloads
```

You're in your home directory, and you want to navigate to your xkcd[45] comic selection in `Documents/my_comics/xkcd`. Instead of typing the full path error-free, you can press Tab after the first few letters. If it is unambiguous, such as `cd Doc <Tab>`, it will expand to `cd Documents`. If there are multiple matching options, such as `cd Do`, you will be prompted for more letters. Pressing Tab again will list the matching options (`Documents` and `Downloads` in this case).

That's it – equipped with the basics of Unix, you are good to go on your DataLad adventure!

---

[45] https://xkcd.com/1205/

WHAT YOU REALLY NEED TO KNOW

DataLad is a data management multitool that can assist you in handling the entire life cycle of digital objects. It is a command-line tool, free and open source, and available for all major operating systems.

This document is the 10.000 feet overview of important concepts, commands, and capacities of DataLad. Each section briefly highlights one type of functionality or concept and the associated commands, and the upcoming Basics chapters will demonstrate in detail how to use them.

5.1 DataLad datasets

Every command affects or uses DataLad *datasets*, the core data structure of DataLad. A *dataset* is a directory on a computer that DataLad manages.

create new, empty datasets to populate...

`% datalad create`

`% datalad create -f`
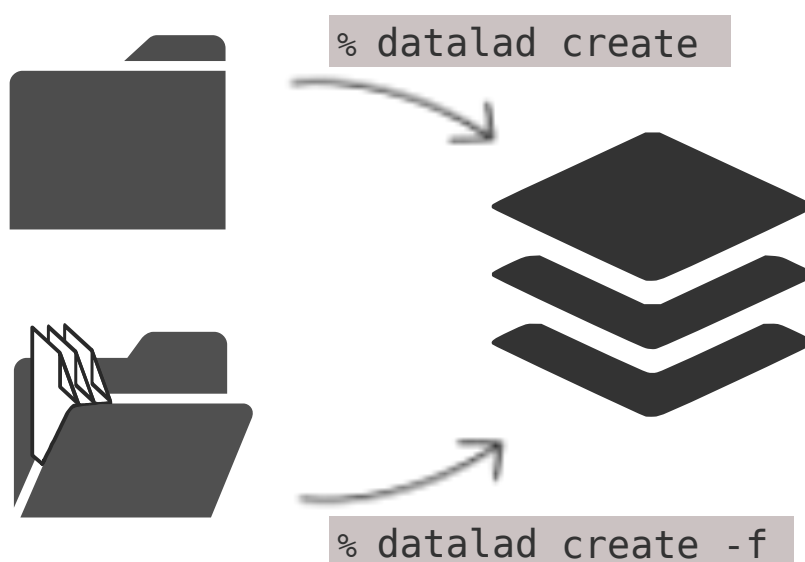
.... or transform existing directories into datasets

You can create new, empty datasets from scratch and populate them, or transform existing directories into datasets.

5.2 Simplified local version control workflows

Building on top of GIT and GIT-ANNEX, DataLad allows you to version control arbitrarily large files in datasets.
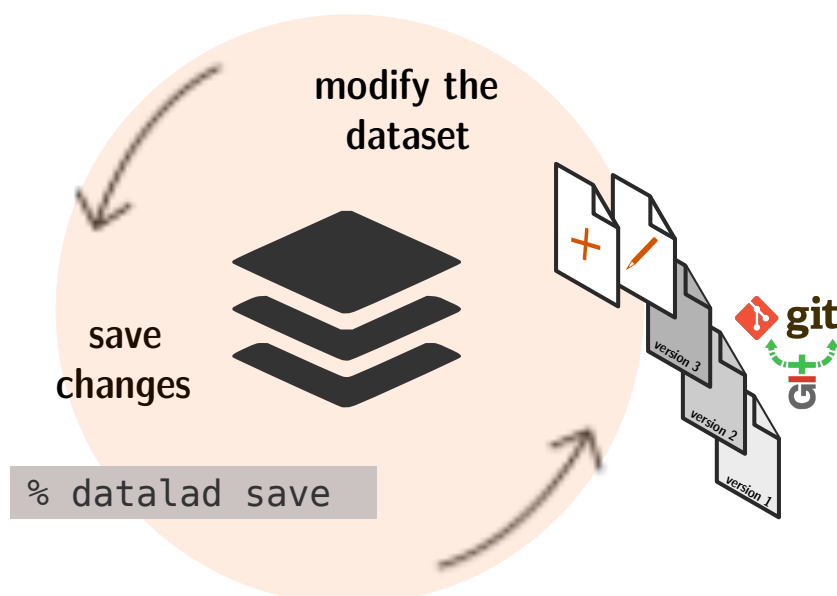
Thus, you can keep track of revisions of data of any size, and view, interact with or restore any version of your dataset's history.

5.3 Consumption and collaboration

DataLad lets you consume datasets provided by others, and collaborate with them. You can install existing datasets and update them from their sources, or create sibling datasets that you can publish updates to and pull updates from for collaboration and data sharing.
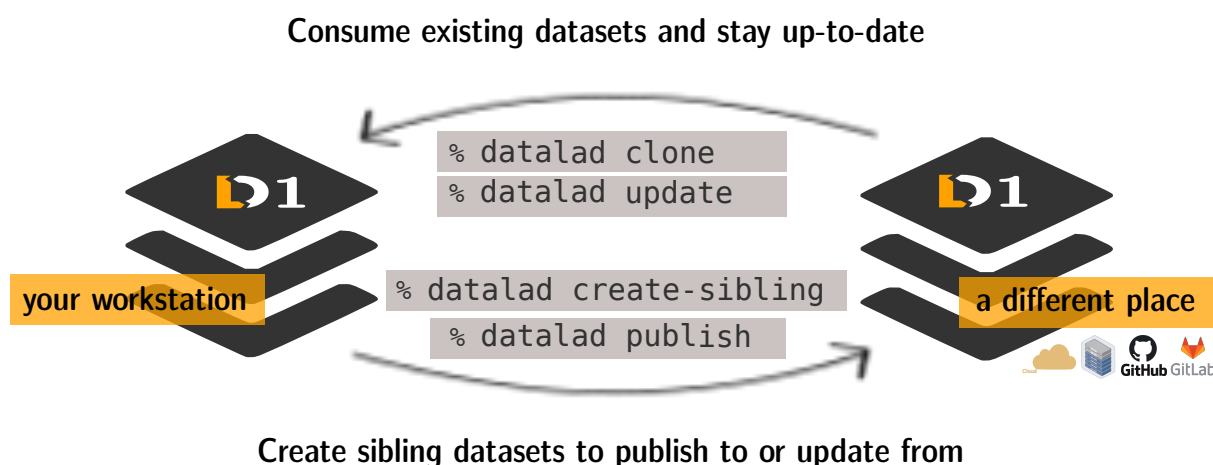
Additionally, you can get access to publicly available open data collections with THE DATALAD SUPERDATASET ///.

5.4 Dataset linkage

Datasets can contain other datasets (subdatasets), nested arbitrarily deep. Each dataset has an independent revision history, but can be registered at a precise version in higher-level datasets. This allows to combine datasets and to perform commands recursively across a hierarchy of datasets, and it is the basis for advanced provenance capture abilities.
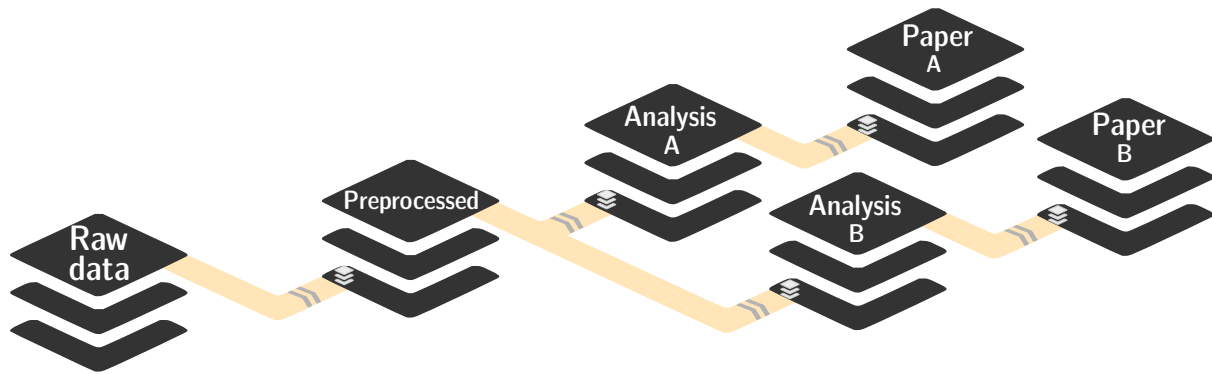
Nest modular datasets to create a linked hierarchy of datasets,
and enable recursive operations throughout the hierarchy

5.5 Full provenance capture and reproducibility

DataLad allows to capture full PROVENANCE: The origin of datasets, the origin of files obtained from web sources, complete machine-readable and automatically reproducible records of how files were created (including software environments).

link input, code, containerized
software environments, and output,
or re-run previous executions

capture the origin
of files obtained
from web sources

```
% datalad download-url
```

```
% datalad run
```

```
% datalad rerun
```

```
% datalad run-procedure
```
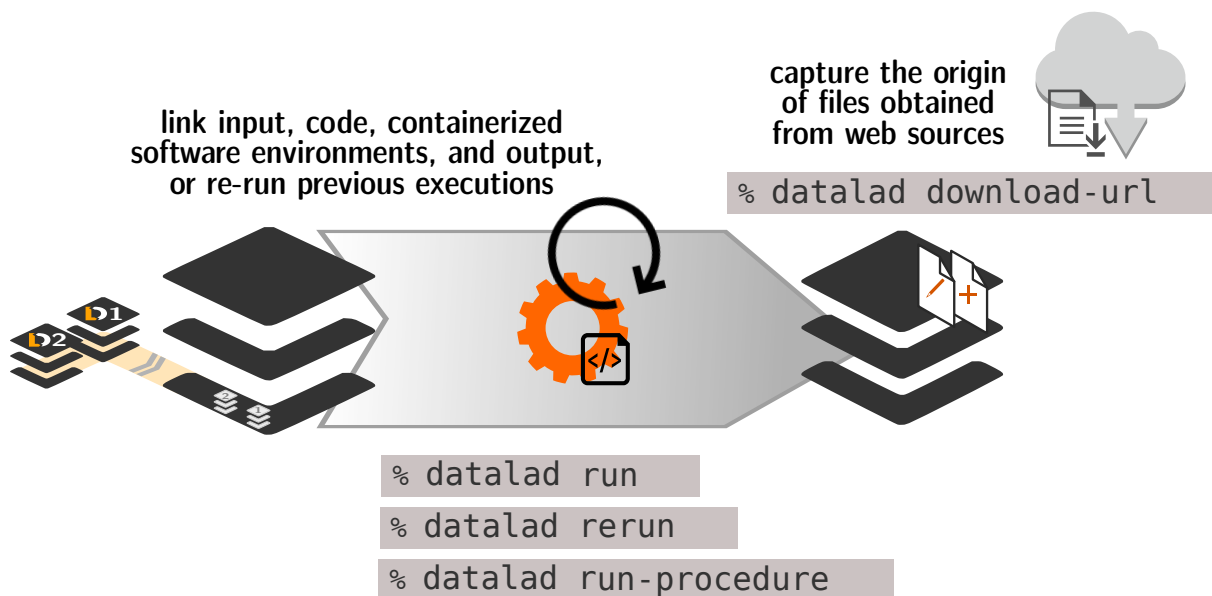
You or your collaborators can thus re-obtain or reproducibly recompute content with a single command, and make use of extensive provenance of dataset content (who created it, when, and how?).

5.6 Third party service integration

Export datasets to third party services such as GitHub[46], GitLab[47], or Figshare[48] with built-in commands.
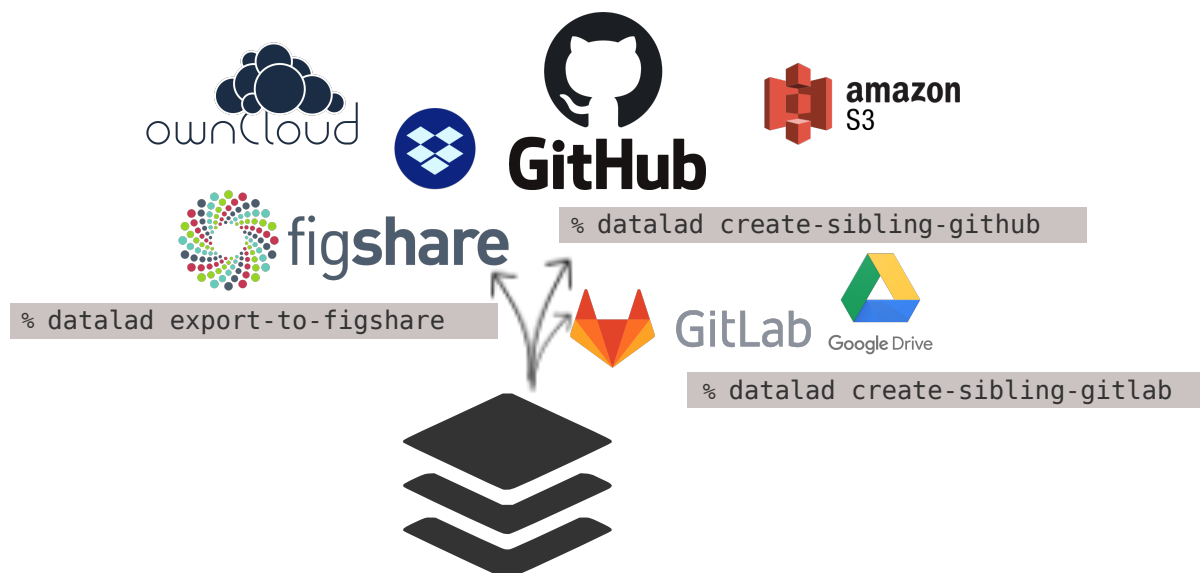
Alternatively, you can use a multitude of other available third party services such as Dropbox[49], Google Drive[50], Amazon S3[51], owncloud[52], or many more that DataLad datasets are compatible with.

5.7 Metadata handling

Extract, aggregate, and query dataset metadata. This allows to automatically obtain metadata according to different metadata standards (EXIF, XMP, ID3, BIDS, DICOM, NIfTI1, . . .), store this metadata in a portable format, share it, and search dataset contents.

---

[46] https://github.com/
[47] https://about.gitlab.com/
[48] https://figshare.com/
[49] https://dropbox.com
[50] https://drive.google.com/drive/my-drive
[51] https://aws.amazon.com/de/s3/
[52] https://owncloud.org/
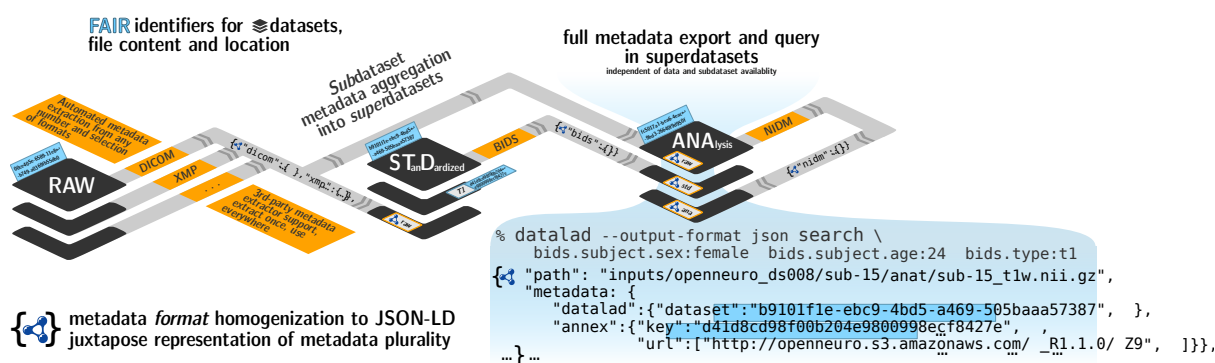
5.8 All in all…

You can use DataLad for a variety of use cases. At its core, it is a domain-agnostic and self-effacing tool: DataLad allows to improve your data management without custom data structures or the need for central infrastructure or third party services. If you are interested in more high-level information on DataLad, you can find answers to common questions in the section *Frequently Asked Questions* (page 479), and a concise command cheat-sheet in section *DataLad cheat sheet* (page 496).

But enough of the introduction now – let's dive into the *Basics* (page 33)
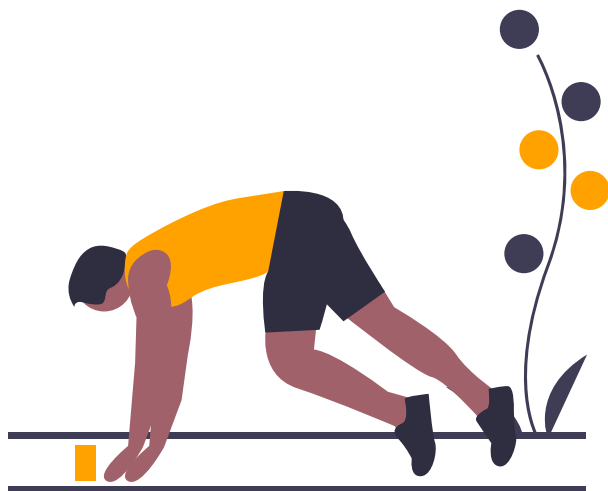
Part II

Basics

The Basics will show you the building blocks of DataLad in a continuous narrative. Start up a terminal, and code along! For the best experience, try reading the Basics chapter sequentially.

DATALAD DATASETS

6.1 Create a dataset

We are about to start the educational course `DataLad-101`. In order to follow along and organize course content, let us create a directory on our computer to collate the materials, assignments, and notes in.

Since this is `DataLad-101`, let's do it as a DATALAD DATASET. You might associate the term "dataset" with a large spreadsheet containing variables and data. But for DataLad, a dataset is the core data type: As noted in *A brief overview of DataLad* (page 2), a dataset is a collection of *files* in folders, and a file is the smallest unit any dataset can contain. Although this is a very simple concept, datasets come with many useful features. Because experiencing is more insightful than just reading, we will explore the concepts of DataLad datasets together by creating one.

Find a nice place on your computer's file system to put a dataset for `DataLad-101`, and create a fresh, empty dataset with the **datalad create** command (datalad-create manual).

Note the command structure of **datalad create** (optional bits are enclosed in [ ]):

```
datalad create [--description "..."] [-c <config options>] PATH
```

> 💡 **M6.1 What is the description option of datalad-create?**
>
> The optional `--description` flag allows you to provide a short description of the *location* of your dataset, for example with

> 💡
> ```
> datalad create --description "course on DataLad-101 on my private Laptop" -c␣
> ↪text2git DataLad-101
> ```
>
> If you want, use the above command instead of the **create** command below to provide
> a description. Its use will not be immediately clear, the chapter *Collaboration* (page 92))
> will show you where this description ends up and how it may be useful.

Let's start:

```
$ datalad create -c text2git DataLad-101
[INFO] Creating a new annex repo at /home/me/dl-101/DataLad-101
[INFO] scanning for unlocked files (this may take some time)
[INFO] Running procedure cfg_text2git
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
create(ok): /home/me/dl-101/DataLad-101 (dataset)
```

This will create a dataset called `DataLad-101` in the directory you are currently in. For now,
disregard `-c text2git`. It applies a configuration template, but there will be other parts of this
book to explain this in detail.

Once created, a DataLad dataset looks like any other directory on your file system. Currently, it
seems empty.

```
$ cd DataLad-101
$ ls     # ls does not show any output, because the dataset is empty.
```

However, all files and directories you store within the DataLad dataset can be tracked (should
you want them to be tracked). *Tracking* in this context means that edits done to a file are
automatically associated with information about the change, the author of the edit, and the time
of this change. This is already informative important on its own – the PROVENANCE captured
with this can for example be used to learn about a file's lineage, and can establish trust in
it. But what is especially helpful is that previous states of files or directories can be restored.
Remember the last time you accidentally deleted content in a file, but only realized *after* you
saved it? With DataLad, no mistakes are forever. We will see many examples of this later in the
book, and such information is stored in what we will refer to as the *history* of a dataset.

This history is almost as small as it can be at the current state, but let's take a look at it. For
looking at the history, the code examples will use **git log**, a built-in GIT command[56] that works
right in your terminal. Your log *might* be opened in a terminal pager[53] that lets you scroll up
and down with your arrow keys, but not enter any more commands. If this happens, you can
get out of git log by pressing q.

```
$ git log
commit a19b634693d18bf507214839520f0031aca371c7
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:24 2021 +0200

    Instruct annex to add text files to Git

commit 9098f563383dce3858c351343e4e671f70bd8243
```

(continues on next page)

---

[56] A tool we can recommend as an alternative to **git log** is TIG. Once installed, exchange any `git log` command
you see here with the single word `tig`.
[53] https://en.wikipedia.org/wiki/Terminal_pager

```
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:23 2021 +0200

    [DATALAD] new dataset
```

We can see two COMMITs in the history of the repository. Each of them is identified by a unique 40 character sequence, called a SHASUM.

> **W6.1 Your Git log may be more extensive - use "git log master" instead!**
>
> The output of `git log` shown in the handbook and the output you will see in your own datasets when executing the same commands may not always match – many times you might see commits about a "git-annex adjusted branch" in your history. This is expected, and if you want to read up more about this, please progress on to chapter 3 and afterwards take a look at this part of git-annex documentation[54].
>
> In order to get a similar experience in your dataset, please add the name of your default BRANCH (it will likely have the name `main` or `master`) to every `git log` command. This should display the same output that the handbook display. The reason behind this is that datasets are using a special BRANCH to be functional on Windows. This branch's history differs from the history that would be in the default branch. With this workaround, you will be able to display the dataset history from the same branch that handbook and all other operating system display. Thus, whenever the handbook code snippet contains a line that starts with `git log`, copy it and append the term `main` or `master`, whichever is appropriate.
>
> If you are eager to help to improve the handbook, you could do us a favor by reporting any places with mismatches between Git logs on Windows and in the handbook. Get in touch[55]!
>
> ---
> [54] https://git-annex.branchable.com/design/adjusted_branches/
> [55] https://github.com/datalad-handbook/book/issues/new/

Highlighted in this output is information about the author and about the time, as well as a COMMIT MESSAGE that summarizes the performed action concisely. In this case, both commit messages were written by DataLad itself. The most recent change is on the top. The first commit written to the history therefore states that a new dataset was created, and the second commit is related to the `-c text2git` option (which uses a configuration template to instruct DataLad to store text files in Git, but more on this later). While these commits were produced and described by DataLad, in most other cases, you will have to create the commit and an informative commit message yourself.

> **G6.1 Create internals**
>
> **datalad create** uses **git init** and **git-annex init**. Therefore, the DataLad dataset is a Git repository. Large file content in the dataset is tracked with git-annex. An `ls -a` reveals that Git has secretly done its work:
>
> ```
> $ ls -a # show also hidden files
> .
> ..
> .datalad
> .git
> .gitattributes
> ```

> **For non-Git-Users: these hidden** *dot-directories* **are necessary for all Git magic to work. Please do not tamper with them, and, importantly,** *do not delete them.*

Congratulations, you just created your first DataLad dataset! Let us now put some content inside.

6.2 Populate a dataset

The first lecture in DataLad-101 referenced some useful literature. Even if we end up not reading those books at all, let's download them nevertheless and put them into our dataset. You never know, right? Let's first create a directory to save books for additional reading in.

```
$ mkdir books
```

Let's take a look at the current directory structure with the tree command[62]:

```
$ tree
.
└── books

1 directory, 0 files
```

Arguably, not the most exciting thing to see. So let's put some PDFs inside. Below is a short list of optional readings. We decide to download them (they are all free, in total about 15 MB), and save them in `DataLad-101/books`.

- Additional reading about the command line: The Linux Command Line[57]
- An intro to Python: A byte of Python[58]

You can either visit the links and save them in books/, or run the following commands[63] to download the books right from the terminal. Note that we line break the command with \ signs. In your own work you can write commands like this into a single line. If you copy them into your terminal as they are presented here, make sure to check the *Windows-wit* 🪟 *W6.2 on peculiarities of its terminals* (page 37).

> **W6.2 Terminals other than Git Bash can't handle multi-line commands**
>
> In Unix shells, \ can be used to split a command into several lines, for example to aid readability. Standard Windows terminals (including the Anaconda prompt) do not support this. They instead use the ^ character:

---

[62] `tree` is a Unix command to list file system content. If it is not yet installed, you can get it with your native package manager (e.g., apt, brew, or conda). For example, if you use OSX, `brew install tree` will get you this tool. On Windows, if you have the Miniconda-based installation described in *Installation and configuration* (page 10), you can install the m2-base package (`conda install m2-base`), which contains tree along with many other Unix-like commands. Note that this tree works slightly different than its Unix equivalent - it will only display directories, not files, and it doesn't accept common options or flags. It will also display *hidden* directories, i.e., those that start with a . (dot).

[57] https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.pdf/download

[58] https://github.com/swaroopch/byte-of-python/releases/download/v14558db59a326ba99eda0da6c4548c48ccb4cd0f/byte-of-python.pdf

[63] `wget` is a Unix command for non-interactively downloading files from the web. If it is not yet installed, you can get it with your native package manager (e.g., apt or brew). For example, if you use OSX, `brew install wget` will get you this tool.

```
$ wget -q https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.
↪pdf/download ^
-O TLCL.pdf
```

If you are not using the Git Bash, you will either need to copy multi-line commands into a single line, or use ^ (make sure that there is **no space** afterwards) instead of \.

```
$ cd books
$ wget -q https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.pdf/
↪download \
  -O TLCL.pdf
$ wget -q https://homepages.uc.edu/~becktl/byte_of_python.pdf \
  -O byte-of-python.pdf
# get back into the root of the dataset
$ cd ../
2021-07-29 16:20:26 URL:https://jztkft.dl.sourceforge.net/project/linuxcommand/TLCL/19.01/
↪TLCL-19.01.pdf [2120211/2120211] -> "TLCL.pdf" [1]
2021-07-29 16:20:27 URL:https://github-releases.githubusercontent.com/6501727/56225300-
↪af61-11ea-8d7f-be2b68e479be?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-
↪Credential=AKIAIWNJYAX4CSVEH53A%2F20210729%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
↪Date=20210729T142026Z&X-Amz-Expires=300&X-Amz-
↪Signature=a9ed817c078f32f88ba17d479828e7496c075bfd27c0528087129b0dd0ca06ee&X-Amz-
↪SignedHeaders=host&actor_id=0&key_id=0&repo_id=6501727&response-content-
↪disposition=attachment%3B%20filename%3Dbyte-of-python.pdf&response-content-
↪type=application%2Foctet-stream [4208954/4208954] -> "byte-of-python.pdf" [1]
```

Some machines will not have `wget` available by default, but any command that can download a file can work as an alternative. See the *Windows-wit* ⊞ *W6.3 for the popular alternative curl* (page 38).

W6.3 You can use curl instead of wget

Many versions of Windows do not ship with the tool `wget`. You can install it, but it may be easier to use the pre-installed `curl` command:

```
$ cd books
$ curl -L https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.
↪pdf/download \
  -o TLCL.pdf
$ curl -L https://homepages.uc.edu/~becktl/byte_of_python.pdf \
  -o byte-of-python.pdf
$ cd ../
```

Let's see what happened. First of all, in the root of `DataLad-101`, show the directory structure with tree:

```
$ tree
.
└── books
    ├── byte-of-python.pdf
    └── TLCL.pdf

1 directory, 2 files
```

Now what does DataLad do with this new content? One command you will use very often

is **datalad status** (datalad-status manual). It reports on the state of dataset content, and regular status reports should become a habit in the wake of DataLad-101.

```
$ datalad status
untracked: books (directory)
```

Interesting; the books/ directory is "untracked". Remember how content *can* be tracked *if a user wants to*? Untracked means that DataLad does not know about this directory or its content, because we have not instructed DataLad to actually track it. This means that DataLad does not store the downloaded books in its history yet. Let's change this by *saving* the files to the dataset's history with the **datalad save** command (datalad-save manual).

This time, it is your turn to specify a helpful COMMIT MESSAGE with the -m option (although the DataLad command is **datalad save**, we talk about commit messages because **datalad save** ultimately uses the command **git commit** to do its work):

```
$ datalad save -m "add books on Python and Unix to read later"
add(ok): books/TLCL.pdf (file)
add(ok): books/byte-of-python.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  save (ok: 1)
```

If you ever forget to specify a message, or made a typo, not all is lost. A *Find-out-more* 💡 *M6.2 explains how to amend a saved state* (page 39).

> **M6.2 "Oh no! I forgot the -m option for datalad-save!"**
>
> If you forget to specify a commit message with the -m option, DataLad will write `[DATALAD] Recorded changes` as a commit message into your history. This is not particularly informative. You can change the *last* commit message with the Git command **git commit --amend**. This will open up your default editor and you can edit the commit message. Careful – the default editor might be VIM! The section *Back and forth in time* (page 245) will show you many more ways in which you can interact with a dataset's history.

As already noted, any files you save in this dataset, and all modifications to these files that you save, are tracked in this history. Importantly, this file tracking works regardless of the size of the files – a DataLad dataset could be your private music or movie collection with single files being many GB in size. This is one aspect that distinguishes DataLad from many other version control tools, among them Git. Large content is tracked in an *annex* that is automatically created and handled by DataLad. Whether text files or larger files change, all of these changes can be written to your DataLad dataset's history.

Let's see how the saved content shows up in the history of the dataset with **git log**. The option -n 1 specifies that we want to take a look at the most recent commit. In order to get a bit more details, we add the -p flag. If you end up in a pager, navigate with up and down arrow keys and leave the log by typing q:

```
$ git log -p -n 1
commit 7ad3b1b54cf743983d2cdb8449ab7968d66bdb1a
Author: Elena Piscopia <elena@example.net>
```

(continues on next page)

> 💡 **M6.3 DOs and DON'Ts for commit messages**
>
> **DOs**
> - Write a *title line* with 72 characters or less (as we did so far)
> - it should be in imperative voice, e.g., "Add notes from lecture 2"
> - Often, a title line is not enough to express your changes and reasoning behind it. In this case, add a body to your commit message by hitting enter twice (before closing the quotation marks), and continue writing a brief summary of the changes after a blank line. This summary should explain "what" has been done and "why", but not "how". Close the quotation marks, and hit enter to save the change with your message.
>
> **DON'Ts**
> - passive voice is hard to read afterwards
> - extensive formatting (hashes, asterisks, quotes, . . .) will most likely make your shell complain
> - it should be obvious: do not say nasty things about other people

```
Date:   Thu Jul 29 16:20:28 2021 +0200

    add books on Python and Unix to read later

diff --git a/books/TLCL.pdf b/books/TLCL.pdf
new file mode 120000
index 0000000..4c84b61
--- /dev/null
+++ b/books/TLCL.pdf
@@ -0,0 +1 @@
+../.git/annex/objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-
↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
\ No newline at end of file
diff --git a/books/byte-of-python.pdf b/books/byte-of-python.pdf
new file mode 120000
index 0000000..adaec61
--- /dev/null
+++ b/books/byte-of-python.pdf
```

Now this might look a bit cryptic (and honestly, tig[64] makes it look prettier). But this tells us the date and time in which a particular author added two PDFs to the directory books/, and thanks to that commit message we have a nice human-readable summary of that action. A *Find-out-more* 💡 *M6.3 explains what makes a good message* (page 40).

> 🔧 **G6.2 There is no staging area in DataLad**
>
> Just as in Git, new files are not tracked from their creation on, but only when explicitly added to Git (in Git terms with an initial `git add`). But different from the common Git workflow, DataLad skips the staging area. A `datalad save` combines a `git add` and a `git commit`, and therefore, the commit message is specified with `datalad save`.

Cool, so now you have added some files to your dataset history. But what is a bit inconvenient is that both books were saved *together*. You begin to wonder: "A Python book and a Unix book do

---

[64] See TIG. Once installed, exchange any git log command you see here with the single word tig.

not have that much in common. I probably should not save them in the same commit. And ...
what happens if I have files I do not want to track? **datalad save -m "some commit message"**
would save all of what is currently untracked or modified in the dataset into the history!"

Regarding your first remark, you're absolutely right! It is good practice to save only those
changes together that belong together. We do not want to squish completely unrelated changes
into the same spot of our history, because it would get very nasty should we want to revert *some*
of the changes without affecting others in this commit.

Luckily, we can point **datalad save** to exactly the changes we want it to record. Let's try this by
adding yet another book, a good reference work about git, Pro Git[59]:

```
$ cd books
$ wget -q https://github.com/progit/progit2/releases/download/2.1.154/progit.pdf
$ cd ../
2021-07-29 16:20:31 URL:https://github-releases.githubusercontent.com/15400220/57552a00-
↪9a49-11e9-9144-d9607ed4c2db?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-
↪Credential=AKIAIWNJYAX4CSVEH53A%2F20210729%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
↪Date=20210729T142029Z&X-Amz-Expires=300&X-Amz-
↪Signature=407a1a69062337b27895dd52a4cbe7424ec23ba0dc02fe2e589e4a9533fa0d86&X-Amz-
↪SignedHeaders=host&actor_id=0&key_id=0&repo_id=15400220&response-content-
↪disposition=attachment%3B%20filename%3Dprogit.pdf&response-content-type=application
↪%2Foctet-stream [12465653/12465653] -> "progit.pdf" [1]
```

datalad status shows that there is a new untracked file:

```
$ datalad status
untracked: books/progit.pdf (file)
```

Let's give **datalad save** precisely this file by specifying its path after the commit message:

```
$ datalad save -m "add reference book about git" books/progit.pdf
add(ok): books/progit.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Regarding your second remark, you're right that a **datalad save** without a path specification
would write all of the currently untracked files or modifications to the history. But check the
Find-out-more 💡 *M6.4 on how to tell it otherwise* (page 41).

> 💡 **M6.4 How to save already tracked dataset components only?**
>
> A **datalad save -m "concise message" --updated** (or the shorter form of --updated,
> -u) will only write *modifications* to the history, not untracked files. Later, we will also
> see .gitignore files that let you hide content from version control. However, it is good
> practice to safely store away modifications or new content. This improves your dataset
> and workflow, and will be a requirement for executing certain commands.

A **datalad status** should now be empty, and our dataset's history should look like this:

---

[59] https://git-scm.com/book/en/v2

```
# lets make the output a bit more concise with the --oneline option
$ git log --oneline
ca8582d add reference book about git
7ad3b1b add books on Python and Unix to read later
a19b634 Instruct annex to add text files to Git
9098f56 [DATALAD] new dataset
```

"Wonderful! I'm getting a hang on this quickly", you think. "Version controlling files is not as hard as I thought!"

But downloading and adding content to your dataset "manually" has two disadvantages: For one, it requires you to download the content and save it. Compared to a workflow with no DataLad dataset, this is one additional command you have to perform (and that additional time adds up, after a while[60]). But a more serious disadvantage is that you have no electronic record of the source of the contents you added. The amount of PROVENANCE, the time, date, and author of file, is already quite nice, but we don't know anything about where you downloaded these files from. If you would want to find out, you would have to *remember* where you got the content from – and brains are not made for such tasks.

Luckily, DataLad has a command that will solve both of these problems: The **datalad download-url** command (datalad-download-url manual). We will dive deeper into the provenance-related benefits of using it in later chapters, but for now, we'll start with best-practice-building. **datalad download-url** can retrieve content from a URL (following any URL-scheme from https, http, or ftp or s3) and save it into the dataset together with a human-readable commit message and a hidden, machine-readable record of the origin of the content. This saves you time, and captures PROVENANCE information about the data you add to your dataset. To experience this, let's add a final book, a beginner's guide to bash[61], to the dataset. We provide the command with a URL, a pointer to the dataset the file should be saved in (. denotes "current directory"), and a commit message.

```
$ datalad download-url \
  http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf \
  --dataset . \
  -m "add beginners guide on bash" \
  -O books/bash_guide.pdf
[INFO] Downloading 'http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf
↪' into '/home/me/dl-101/DataLad-101/books/bash_guide.pdf'
download_url(ok): /home/me/dl-101/DataLad-101/books/bash_guide.pdf (file)
add(ok): books/bash_guide.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
```

Afterwards, a fourth book is inside your books/ directory:

```
$ ls books
bash_guide.pdf
byte-of-python.pdf
progit.pdf
TLCL.pdf
```

---

[60] https://xkcd.com/1205/
[61] https://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

However, the **datalad status** command does not return any output – the dataset state is "clean":

```
$ datalad status
nothing to save, working tree clean
```

This is because **datalad download-url** took care of saving for you:

```
$ git log -p -n 1
commit 19306998c57dc14e0d735a2a50512905c33c5779
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:36 2021 +0200

    add beginners guide on bash

diff --git a/books/bash_guide.pdf b/books/bash_guide.pdf
new file mode 120000
index 0000000..00ca6bd
--- /dev/null
+++ b/books/bash_guide.pdf
@@ -0,0 +1 @@
+../.git/annex/objects/WF/Gq/MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-
↪s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
\ No newline at end of file
```

At this point in time, the biggest advantage may seem to be the time save. However, soon you will experience how useful it is to have DataLad keep track for you where file content came from.

To conclude this section, let's take a final look at the history of your dataset at this point:

```
$ git log --oneline
1930699 add beginners guide on bash
ca8582d add reference book about git
7ad3b1b add books on Python and Unix to read later
a19b634 Instruct annex to add text files to Git
9098f56 [DATALAD] new dataset
```

Well done! Your DataLad-101 dataset and its history are slowly growing.

6.3 Modify content

So far, we've only added new content to the dataset. And we have not done much to that content up to this point, to be honest. Let's see what happens if we add content, and then modify it.

For this, in the root of DataLad-101, create a plain text file called notes.txt. It will contain all of the notes that you take throughout the course.

Let's write a short summary of how to create a DataLad dataset from scratch:

> "One can create a new dataset with 'datalad create [--description] PATH'. The dataset is created empty".

This is meant to be a note you would take in an educational course. You can take this note and write it to a file with an editor of your choice. The code below, however, contains this note

within the start and end part of a here document[65]. You can also copy the full code snippet, starting from cat << EOT > notes.txt, including the EOT in the last line, in your terminal to write this note from the terminal (without any editor) into notes.txt.

M6.5 How does a here-document work?

The code snippet below makes sure to write lines of text into a file (that so far does not exist) called notes.txt.

To do this, the content of the "document" is wrapped in between *delimiting identifiers*. Here, these identifiers are *EOT* (short for "end of text"), but naming is arbitrary as long as the two identifiers are identical. The first "EOT" identifies the start of the text stream, and the second "EOT" terminates the text stream.

The characters << redirect the text stream into "standard input" (stdin)[66], the standard location that provides the *input* for a command. Thus, the text stream becomes the input for the cat command[67], which takes the input and writes it to "standard output" (stdout)[68].

Lastly, the > character takes stdout can creates a new file notes.txt with stdout as its contents.

It might seem like a slightly convoluted way to create a text file with a note in it. But it allows to write notes from the terminal, enabling this book to create commands you can execute with nothing other than your terminal. You are free to copy-paste the snippets with the here-documents, or find a workflow that suites you better. The only thing important is that you create and modify a .txt file over the course of the Basics part of this handbook.

---

[66] https://en.wikipedia.org/wiki/Standard_streams#Standard_input_(stdin)
[67] https://en.wikipedia.org/wiki/Cat_(Unix)
[68] https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)

Running the command below will create notes.txt in the root of your DataLad-101 dataset:

W6.4 Heredocs don't work under non-Git-Bash Windows terminals

Heredocs rely on Unix-type redirection and multi-line commands – which is not supported on most native Windows terminals or the Anaconda prompt on Windows. If you are using an Anaconda prompt or a Windows terminal other than Git Bash, instead of executing heredocs, please open up an editor and paste and save the text into it.

The relevant text in the snippet below would be:

```
One can create a new dataset with 'datalad create [--description] PATH'.
The dataset is created empty
```

If you are using Git Bash, however, here docs will work just fine.

```
$ cat << EOT > notes.txt
One can create a new dataset with 'datalad create [--description] PATH'.
The dataset is created empty

EOT
```

Run **datalad status** to confirm that there is a new, untracked file:

---

[65] https://en.wikipedia.org/wiki/Here_document

```
$ datalad status
untracked: notes.txt (file)
```

Save the current state of this file in your dataset's history. Because it is the only modification in the dataset, there is no need to specify a path.

```
$ datalad save -m "Add notes on datalad create"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

But now, let's see how *changing* tracked content works. Modify this file by adding another note. After all, you already know how to use **datalad save**, so write a short summary on that as well.

Again, the example below uses Unix commands (cat and redirection, this time however with >> to *append* new content to the existing file) to accomplish this, but you can take any editor of your choice.

```
$ cat << EOT >> notes.txt
The command "datalad save [-m] PATH" saves the file (modifications) to
history.
Note to self: Always use informative, concise commit messages.

EOT
```

Let's check the dataset's current state:

```
$ datalad status
 modified: notes.txt (file)
```

and save the file in DataLad:

```
$ datalad save -m "add note on datalad save"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Let's take another look into our history to see the development of this file. We're using **git log -p -n 2** to see last two commits and explore the difference to the previous state of a file within each commit.

```
$ git log -p -n 2
commit 13cc7f73f0ff5054aef855aebcceca203f98ed09
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:41 2021 +0200

    add note on datalad save

diff --git a/notes.txt b/notes.txt
index 3a7a1fe..0142412 100644
--- a/notes.txt
+++ b/notes.txt
```

(continues on next page)

```
@@ -1,3 +1,7 @@
 One can create a new dataset with 'datalad create [--description] PATH'.
 The dataset is created empty

+The command "datalad save [-m] PATH" saves the file (modifications) to
+history.
+Note to self: Always use informative, concise commit messages.
+

commit cc213c2d401c467f3944559bc97acb1eecd026fe
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:40 2021 +0200

    Add notes on datalad create

diff --git a/notes.txt b/notes.txt
new file mode 100644
```

We can see that the history can not only show us the commit message attached to a commit, but also the precise change that occurred in the text file in the commit. Additions are marked with a +, and deletions would be shown with a leading -. From the dataset's history, we can therefore also find out *how* the text file evolved over time. That's quite neat, isn't it?

> **M6.6 git log has many more useful options**
>
> git log, as many other Git commands, has a good number of options which you can discover if you run git log --help. Those options could help to find specific changes (e.g., which added or removed a specific word with -S), or change how git log output will look (e.g., --word-diff to highlight individual word changes in the -p output).

6.4 Install datasets

So far, we have created a DataLad-101 course dataset. We saved some additional readings into the dataset, and have carefully made and saved notes on the DataLad commands we discovered. Up to this point, we therefore know the typical, *local* workflow to create and populate a dataset from scratch.

But we've been told that with DataLad we could very easily get vast amounts of data to our computer. Rumor has it that this would be only a single command in the terminal! Therefore, everyone in today's lecture excitedly awaits today's topic: Installing datasets.

"With DataLad, users can install *clones* of existing DataLad datasets from paths, URLs, or open-data collections" our lecturer begins. "This makes accessing data fast and easy. A dataset that others could install can be created by anyone, without a need for additional software. Your own datasets can be installed by others, should you want that, for example. Therefore, not only accessing data becomes fast and easy, but also *sharing*." "That's so cool!", you think. "Exam preparation will be a piece of cake if all of us can share our mid-term and final projects easily!" "But today, let's only focus on how to install a dataset", she continuous. "Damn it! Can we not have longer lectures?", you think and set alarms to all of the upcoming lecture dates in your calendar. There is so much exciting stuff to come, you can not miss a single one.

"Psst!" a student from the row behind reaches over. "There are a bunch of audio recordings of

a really cool podcast, and they have been shared in the form of a DataLad dataset! Shall we try whether we can install that?"

"Perfect! What a great way to learn how to install a dataset. Doing it now instead of looking at slides for hours is my preferred type of learning anyway", you think as you fire up your terminal and navigate into your DataLad-101 dataset.

In this demonstration, we're using one of the many openly available datasets that DataLad provides in a public registry that anyone can access. One of these datasets is a collection of audio recordings of a great podcast, the longnow seminar series[71]. It consists of audio recordings about long-term thinking, and while the DataLad-101 course is not a long-term thinking seminar, those recordings are nevertheless a good addition to the large stash of yet-to-read text books we piled up. Let's get this dataset into our existing DataLad-101 dataset.

To keep the DataLad-101 dataset neat and organized, we first create a new directory, called recordings.

```
# we are in the root of DataLad-101
$ mkdir recordings
```

There are two commands that can be used to obtain a dataset: **datalad install** (datalad-install manual) and **datalad clone** (datalad-clone manual). Throughout this handbook, we will use **datalad clone** to obtain datasets. The command has a less complex structure but slightly simplified behavior, and *the Findoutmore* (page 98) in section *Looking without touching* (page 92) will elaborate on the differences between the two commands. Let's install the longnow podcasts in this new directory with **datalad clone**.

The command takes a location of an existing dataset to clone. This *source* can be a URL or a path to a local directory, or an SSH server[70]. The dataset to be installed lives on GITHUB, at https://github.com/datalad-datasets/longnow-podcasts.git[69], and we can give its GitHub URL as the first positional argument. Optionally, the command also takes as second positional argument a path to the *destination*, – a path to where we want to install the dataset to. In this case it is recordings/longnow. Because we are installing a dataset (the podcasts) into an existing dataset (the DataLad-101 dataset), we also supply a -d/--dataset flag to the command. This specifies the dataset to perform the operation on, and allows us to install the podcasts as a *subdataset* of DataLad-101. Because we are in the root of the DataLad-101 dataset, the pointer to the dataset is a . (which is Unix' way of saying "current directory").

As before with long commands, we line break the code below with a \. You can copy it as it is presented here into your terminal, but in your own work you can write commands like this into a single line.

```
$ datalad clone --dataset . \
 https://github.com/datalad-datasets/longnow-podcasts.git recordings/longnow
[INFO] Cloning dataset to Dataset(/home/me/dl-101/DataLad-101/recordings/longnow)
[INFO] Attempting to clone from https://github.com/datalad-datasets/longnow-podcasts.git↵
↪to /home/me/dl-101/DataLad-101/recordings/longnow
[INFO] Start enumerating objects
```

(continues on next page)

---

[71] The longnow podcasts are lectures and conversations on long-term thinking produced by the LongNow foundation and we can wholeheartedly recommend them for their worldly wisdoms and compelling, thoughtful ideas. Subscribe to the podcasts at https://longnow.org/seminars/podcast. Support the foundation by becoming a member: https://longnow.org/membership. https://longnow.org

[70] Additionally, a source can also be a pointer to an open-data collection, for example THE DATALAD SUPERDATASET /// – more on what this is and how to use it later, though.

[69] https://github.com/datalad-datasets/longnow-podcasts

```
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/DataLad-101/recordings/
↪longnow)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Remote origin not usable by git-annex; setting annex-ignore
install(ok): recordings/longnow (dataset)
add(ok): recordings/longnow (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

This command copied the repository found at the URL https://github.com/datalad-datasets/
longnow-podcasts.git into the existing `DataLad-101` dataset, into the directory `recordings/`
`longnow`. The optional destination is helpful: If we had not specified the path `recordings/`
`longnow` as a destination for the dataset clone, the command would have installed the dataset
into the root of the `DataLad-101` dataset, and instead of `longnow` it would have used the name
of the remote repository "`longnow-podcasts`". But the coolest feature of **`datalad clone`** is yet
invisible: This command also recorded where this dataset came from, thus capturing its *origin*
as PROVENANCE. Even though this is not obvious at this point in time, later chapters in this
handbook will demonstrate how useful this information can be.

> **M6.7 Do I have to install from the root of datasets?**
>
> No. Instead of from the *root* of the `DataLad-101` dataset, you could have also installed
> the dataset from within the `recordings`, or `books` directory. In the case of installing
> datasets into existing datasets you however need to adjust the paths that are given with
> the `-d/--dataset` option: `-d` needs to specify the path to the root of the dataset. This
> is important to keep in mind whenever you do not execute the **`clone`** command from
> the root of this dataset. Luckily, there is a shortcut: `-d^` will always point to root of the
> top-most dataset. For example, if you navigate into `recordings` the command would be:
>
> ```
> datalad clone -d^ https://github.com/datalad-datasets/longnow-podcasts.git longnow
> ```

> **M6.8 What if I do not install into an existing dataset?**
>
> If you do not install into an existing dataset, you only need to omit the `-d/--dataset`
> option. You can try:
>
> ```
> datalad clone https://github.com/datalad-datasets/longnow-podcasts.git
> ```
>
> anywhere outside of your `DataLad-101` dataset to install the podcast dataset into a new
> directory called `longnow-podcasts`. You could even do this inside of an existing dataset.
> However, whenever you install datasets into of other datasets, the `-d/--dataset` option is
> necessary to not only install the dataset, but also *register* it automatically into the higher
> level *superdataset*. The upcoming section will elaborate on this.

> **G6.3 Clone internals**
>
> The **datalad clone** command uses **git clone**. A dataset that is installed from an existing source, e.g., a path or URL, is the DataLad equivalent of a *clone* in Git.

Here is the repository structure:

> **W6.5 tree -d may fail**
>
> If you have installed CONDAs m2-base package for access to Unix commands such as tree, you will have the tree command. However, this version of tree does not support the use of any command flags, so please just run tree instead of tree -d.

```
$ tree -d   # we limit the output to directories
.
├── books
└── recordings
    └── longnow
        ├── Long_Now__Conversations_at_The_Interval
        └── Long_Now__Seminars_About_Long_term_Thinking

5 directories
```

We can see that recordings has one subdirectory, our newly installed longnow dataset. Within the dataset are two other directories, Long_Now__Conversations_at_The_Interval and Long_Now__Seminars_About_Long_term_Thinking. If we navigate into one of them and list its content, we'll see many .mp3 files (here is an excerpt).

```
$ cd recordings/longnow/Long_Now__Seminars_About_Long_term_Thinking
$ ls
2003_11_15__Brian_Eno__The_Long_Now.mp3
2003_12_13__Peter_Schwartz__The_Art_Of_The_Really_Long_View.mp3
2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_
↪Large_scale_Computing.mp3
2004_02_14__James_Dewar__Long_term_Policy_Analysis.mp3
2004_03_13__Rusty_Schweickart__The_Asteroid_Threat_Over_the_Next_100_000_Years.mp3
2004_04_10__Daniel_Janzen__Third_World_Conservation__It_s_ALL_Gardening.mp3
2004_05_15__David_Rumsey__Mapping_Time.mp3
2004_06_12__Bruce_Sterling__The_Singularity__Your_Future_as_a_Black_Hole.mp3
2004_07_10__Jill_Tarter__The_Search_for_Extra_terrestrial_Intelligence__Necessarily_a_
↪Long_term_Strategy.mp3
2004_08_14__Phillip_Longman__The_Depopulation_Problem.mp3
2004_09_11__Danny_Hillis__Progress_on_the_10_000_year_Clock.mp3
2004_10_16__Paul_Hawken__The_Long_Green.mp3
2004_11_13__Michael_West__The_Prospects_of_Human_Life_Extension.mp3
```

Dataset content identity and availability information

Surprised, you turn to your fellow student and wonder about how fast the dataset was installed. Should a download of that many .mp3 files not take much more time?

Here you can see another import feature of DataLad datasets and the **datalad clone** command: Upon installation of a DataLad dataset, DataLad retrieves only small files (for example text files or markdown files) and (small) metadata information about the dataset. It does not, however,

download any large files (yet). The metadata exposes the dataset's file hierarchy for exploration (note how you are able to list the dataset contents with ls), and downloading only this metadata speeds up the installation of a DataLad dataset of many TB in size to a few seconds. Just now, after installing, the dataset is small in size:

```
$ cd ../      # in longnow/
$ du -sh      # Unix command to show size of contents
3.7M        .
```

This is tiny indeed!

If you executed the previous ls command in your own terminal, you might have seen the .mp3 files highlighted in a different color than usually. On your computer, try to open one of the .mp3 files. You will notice that you cannot open any of the audio files. This is not your fault: *None of these files exist on your computer yet*.

Wait, what?

This sounds strange, but it has many advantages. Apart from a fast installation, it allows you to retrieve precisely the content you need, instead of all the contents of a dataset. Thus, even if you install a dataset that is many TB in size, it takes up only few MB of space after the install, and you can retrieve only those components of the dataset that you need.

Let's see how large the dataset would be in total if all of the files were present. For this, we supply an additional option to **datalad status**. Make sure to be (anywhere) inside of the longnow dataset to execute the following command:

```
$ datalad status --annex
236 annex'd files (15.4 GB recorded total size)
nothing to save, working tree clean
```

Woah! More than 200 files, totaling more than 15 GB? You begin to appreciate that DataLad did not download all of this data right away! That would have taken hours given the crappy internet connection in the lecture hall, and you are not even sure whether your hard drive has much space left...

But you nevertheless are curious on how to actually listen to one of these .mp3s now. So how does one actually "get" the files?

The command to retrieve file content is **datalad get** (datalad-get manual). You can specify one or more specific files, or get all of the dataset by specifying **datalad get .** (with . denoting "current directory").

First, we get one of the recordings in the dataset – take any one of your choice (here, its the first).

```
$ datalad get Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_
↪Now.mp3
get(ok): Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.
↪mp3 (file) [from web...]
```

Try to open it – it will now work.

If you would want to get the rest of the missing data, instead of specifying all files individually, we can use . to refer to *all* of the dataset like this:

```
$ datalad get .
```

However, with a total size of more than 15GB, this might take a while, so do not do that now. If you did execute the command above, interrupt it by pressing CTRL + C – Do not worry, this will not break anything.

Isn't that easy? Let's see how much content is now present locally. For this, **datalad status --annex all** has a nice summary:

```
$ datalad status --annex all
236 annex'd files (35.7 MB/15.4 GB present/total size)
nothing to save, working tree clean
```

This shows you how much of the total content is present locally. With one file, it is only a fraction of the total size.

Let's get a few more recordings, just because it was so mesmerizing to watch DataLad's fancy progress bars.

```
$ datalad get Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_
↪Now.mp3 \
Long_Now__Seminars_About_Long_term_Thinking/2003_12_13__Peter_Schwartz__The_Art_Of_The_
↪Really_Long_View.mp3 \
Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_Plenty_of_
↪Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3
get(ok): Long_Now__Seminars_About_Long_term_Thinking/2003_12_13__Peter_Schwartz__The_Art_
↪Of_The_Really_Long_View.mp3 (file) [from web...]
get(ok): Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_
↪Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3 (file)↵
↪[from web...]
action summary:
  get (notneeded: 1, ok: 2)
```

Note that any data that is already retrieved (the first file) is not downloaded again. DataLad summarizes the outcome of the execution of get in the end and informs that the download of one file was notneeded and the retrieval of the other files was ok.

> **G6.4 Get internals**
>
> **datalad get** uses **git annex get** underneath the hood.

Keep whatever you like

"Oh shit, oh shit, oh shit. . . " you hear from right behind you. Your fellow student apparently downloaded the *full* dataset accidentally. "Is there a way to get rid of file contents in dataset, too?", they ask. "Yes", the lecturer responds, "you can remove file contents by using **datalad drop**. This is really helpful to save disk space for data you can easily re-obtain, for example".

The **datalad drop** command (datalad drop manual) will remove file contents completely from your dataset. You should only use this command to remove contents that you can **get** again, or generate again (for example with next chapter's **datalad run** command), or that you really do not need anymore.

Let's remove the content of one of the files that we have downloaded, and check what this does to the total size of the dataset. Here is the current amount of retrieved data in this dataset:

```
$ datalad status --annex all
236 annex'd files (135.1 MB/15.4 GB present/total size)
nothing to save, working tree clean
```

We drop a single recording that's content we previously downloaded with **get** . . .

```
$ datalad drop Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__
↪There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3
drop(ok): /home/me/dl-101/DataLad-101/recordings/longnow/Long_Now__Seminars_About_Long_
↪term_Thinking/2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_
↪Thinking_About_Large_scale_Computing.mp3 (file) [checking http://podcast.longnow.org/
↪salt/redirect/salt-020040109-dyson-podcast.mp3...]
```

. . . and check the size of the dataset again:

```
$ datalad status --annex all
236 annex'd files (93.5 MB/15.4 GB present/total size)
nothing to save, working tree clean
```

Dropping the file content of one mp3 file saved roughly 40MB of disk space. Whenever you need the recording again, it is easy to re-retrieve it:

```
$ datalad get Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_
↪s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3
get(ok): Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_
↪Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3 (file)␣
↪[from web...]
```

Re-obtained!

This was only a quick digression into **datalad drop**. The main principles of this command will become clear after chapter *Under the hood: git-annex* (page 83), and its precise use is shown in the paragraph on removing file contents. At this point in time, however, you already know that datasets allow you do **drop** file contents flexibly. If you want to, you could have more podcasts (or other data) on your computer than you have disk space available by using DataLad datasets – and that really is a cool feature to have.

Dataset archeology

You have now experienced how easy it is to (re-)obtain shared data with DataLad. But beyond only sharing the *data* in the dataset, when sharing or installing a DataLad dataset, all copies also include the datasets *history*.

For example, we can find out who created the dataset in the first place (the output shows an excerpt of git log --reverse, which displays the history from first to most recent commit):

```
$ git log --reverse
commit 8df130bb825f99135c34b8bf0cbedb1b05edd581
Author: Michael Hanke <michael.hanke@gmail.com>
Date:   Mon Jul 16 16:08:23 2018 +0200

    [DATALAD] Set default backend for all files to be MD5E

commit 3d0dc8f5e9e4032784bc5a08d243995ad5cf92f9
```

(continues on next page)

```
Author: Michael Hanke <michael.hanke@gmail.com>
Date:   Mon Jul 16 16:08:24 2018 +0200

    [DATALAD] new dataset
```

But that's not all. The seminar series is ongoing, and more recordings can get added to the original repository shared on GitHub. Because an installed dataset knows the dataset it was installed from, your local dataset clone can be updated from its origin, and thus get the new recordings, should there be some. Later in this handbook, we will see examples of this.

Now you can not only create datasets and work with them locally, you can also consume existing datasets by installing them. Because that's cool, and because you will use this command frequently, make a note of it into your `notes.txt`, and **datalad save** the modification.

```
# in the root of DataLad-101:
$ cd ../../
$ cat << EOT >> notes.txt
The command 'datalad clone URL/PATH [PATH]' installs a dataset from
e.g., a URL or a path. If you install a dataset into an existing
dataset (as a subdataset), remember to specify the root of the
superdataset with the '-d' option.

EOT
$ datalad save -m "Add note on datalad clone"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

> ⚠️ **Empty files can be confusing**
>
> Listing files directly after the installation of a dataset will work if done in a terminal with `ls`. However, certain file managers (such as OSX's Finder[72]) may fail to display files that are not yet present locally (i.e., before a **datalad get** was run). Therefore, be mindful when exploring a dataset hierarchy with a file manager – it might not show you the available but not yet retrieved files. More about why this is will be explained in section *Data integrity* (page 85).
>
> ---
> [72] You can also upgrade your file manager to display file types in a DataLad datasets (e.g., with the git-annex-turtle extension[73] for Finder)

6.5 Dataset nesting

Without noticing, the previous section demonstrated another core principle and feature of DataLad datasets: *Nesting*.

Within DataLad datasets one can *nest* other DataLad datasets arbitrarily deep. We for example just installed one dataset, the `longnow` podcasts, *into* another dataset, the `DataLad-101` dataset. This was done by supplying the `--dataset/-d` flag in the command call.

At first glance, nesting does not seem particularly spectacular – after all, any directory on a file system can have other directories inside of it.

The possibility for nested Datasets, however, is one of many advantages DataLad datasets have:

One aspect of nested datasets is that any lower-level DataLad dataset (the *subdataset*) has a stand-alone history. The top-level DataLad dataset (the *superdataset*) only stores *which version* of the subdataset is currently used.

Let's dive into that. Remember how we had to navigate into `recordings/longnow` to see the history, and how this history was completely independent of the `DataLad-101` superdataset history? This was the subdataset's own history.

Apart from stand-alone histories of super- or subdatasets, this highlights another very important advantage that nesting provides: Note that the `longnow` dataset is a completely independent, standalone dataset that was once created and published. Nesting allows for a modular re-use of any other DataLad dataset, and this re-use is possible and simple precisely because all of the information is kept within a (sub)dataset.

But now let's also check out how the *superdataset's* (`DataLad-101`) history looks like after the addition of a subdataset. To do this, make sure you are *outside* of the subdataset `longnow`. Note that the first commit is our recent addition to `notes.txt`, so we'll look at the second most recent commit in this excerpt.

```
$ git log -p -n 3
commit 6cd43dd1c70c480a9745d7bfdfdfdbcbc4b32f43
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:47 2021 +0200

    [DATALAD] modified subdataset properties

diff --git a/.gitmodules b/.gitmodules
index 1b59b8c..9bc9ee9 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -2,3 +2,4 @@
        path = recordings/longnow
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
+       datalad-url = https://github.com/datalad-datasets/longnow-podcasts.git

commit 24722102f8ebfd5857298142cb2d3846a8339359
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:20:47 2021 +0200

    [DATALAD] Recorded changes

diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..1b59b8c
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,4 @@
+[submodule "recordings/longnow"]
+       path = recordings/longnow
+       url = https://github.com/datalad-datasets/longnow-podcasts.git
+       datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
diff --git a/recordings/longnow b/recordings/longnow
new file mode 160000
index 0000000..dcc34fb
```

```
--- /dev/null
+++ b/recordings/longnow
@@ -0,0 +1 @@
+Subproject commit dcc34fbe669b06ced84ced381ba0db21cf5e665f

commit 13cc7f73f0ff5054aef855aebccceca203f98ed09
```

We have highlighted the important part of this rather long commit summary. Note that you can not see any `.mp3s` being added to the dataset, as was previously the case when we **datalad save**d PDFs that we downloaded into `books/`. Instead, DataLad stores what it calls a *subproject commit* of the subdataset. The cryptic character sequence in this line is the SHASUM we have briefly mentioned before, and it is how DataLad internally identifies files and changes to files. Exactly this SHASUM is what describes the state of the subdataset.

Navigate back into `longnow` and try to find the highlighted shasum in the subdataset's history:

```
$ cd recordings/longnow
$ git log --oneline
dcc34fb Update aggregated metadata
36a30a1 [DATALAD RUNCMD] Update from feed
bafdc04 Uniformize JSON-LD context with DataLad's internal extractors
004e484 [DATALAD RUNCMD] .datalad/maint/make_readme.py
7ee3ded Sort episodes newest-first
e829615 Link to the handbook as a source of wisdom
4b37790 Fix README generator to parse correct directory
```

We can see that it is the most recent commit shasum of the subdataset (albeit we can see only the first seven characters here – a **git log** would show you the full shasum). Thus, your dataset does not only know the origin of its subdataset, but also its version, i.e., it has an identifier of the stage of the subdatasets evolution. This is what is meant by "the top-level DataLad dataset (the *superdataset*) only stores *which version* of the subdataset is currently used".

Importantly, once we learn how to make use of the history of a dataset, we can set subdatasets to previous states, or *update* them.

> 💡 **M6.9 Do I have to navigate into the subdataset to see it's history?**
>
> Previously, we used **cd** to navigate into the subdataset, and subsequently opened the Git log. This is necessary, because a **git log** in the superdataset would only return the superdatasets history. While moving around with cd is straightforward, you also found it slightly annoying from time to time to use the cd command so often and also to remember in which directory you currently are in. There is one trick, though: `git -C` (note that it is a capital C) lets you perform any Git command in a provided path. Providing this option together with a path to a Git command let's you run the command as if Git was started in this path instead of the current working directory. Thus, from the root of `DataLad-101`, this command would have given you the subdataset's history as well:
>
> ```
> $ git -C recordings/longnow log --oneline
> ```

In the upcoming sections, we'll experience the perks of dataset nesting frequently, and everything that might seem vague at this point will become clearer. To conclude this demonstration, the figure below illustrates the current state of the dataset and nesting schematically:

Thus, without being consciously aware of it, by taking advantage of dataset nesting, we took a
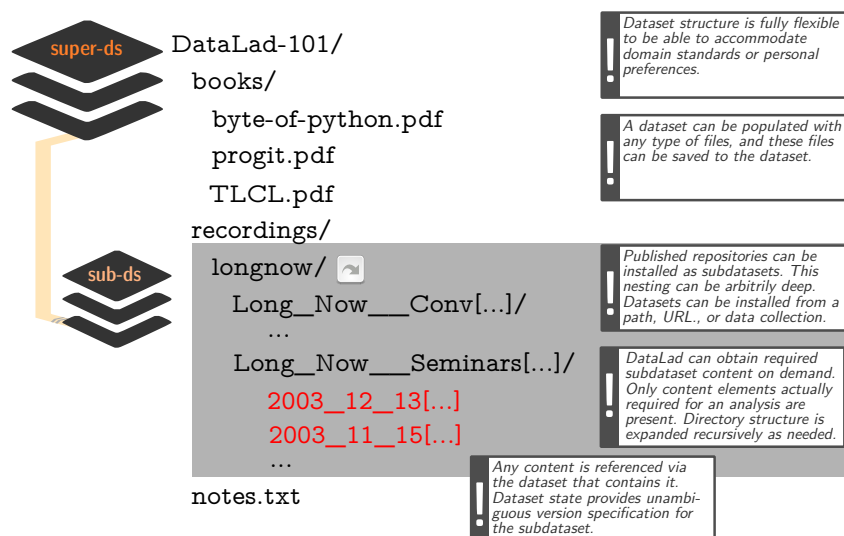
Fig. 6.1: Virtual directory tree of a nested DataLad dataset

dataset `longnow` and installed it as a subdataset within the superdataset `DataLad-101`.

If you have executed the above code snippets, make sure to go back into the root of the dataset again:

```
$ cd ../../
```

6.6 Summary

In the last few sections, we have discovered the basics of starting a DataLad dataset from scratch, and making simple modifications *locally*.

- An empty dataset can be created with the **datalad create** command. It's useful to add a description to the dataset and use the `-c text2git` configuration, but we will see later why. This is the command structure:

```
datalad create --description "here is a description" -c text2git PATH
```

- Thanks to GIT and GIT-ANNEX, the dataset has a history to track files and their modifications. Built-in Git tools (**git log**) or external tools (such as `tig`) allow to explore the history.

- The **datalad save** command records the current state of the dataset to the history. Make it a habit to specify a concise commit message to summarize the change. If several unrelated modifications exist in your dataset, specify the path to the precise file (change) that should be saved to history. Remember, if you run a **datalad save** without specifying a path, all untracked files and all file changes will be committed to the history together! This is the command structure:

```
datalad save -m "here is a commit message" [PATH]
```

- The typical local workflow is simple: *Modify* the dataset by adding or modifying files, *save* the changes as meaningful units to the history, *repeat*:
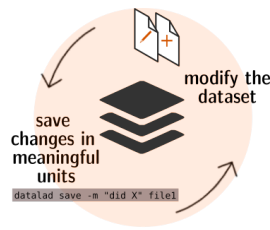
Fig. 6.2: A simple, local version control workflow with DataLad.

- **datalad status** reports the current state of the dataset. It's a very helpful command you should run frequently to check for untracked or modified content.

- **datalad download-url** can retrieve files from websources and save them automatically to your dataset. This does not only save you the time of one **datalad save**, but it also records the source of the file as hidden PROVENANCE information.

Furthermore, we have discovered the basics of installing a published DataLad dataset, and experienced the concept of modular nesting datasets.

- A published dataset can be installed with the **datalad clone** command:

  ```
  $ datalad clone [--dataset PATH] SOURCE-PATH/URL [DESTINATION PATH]
  ```

  It can be installed "on its own", or within an existing dataset.

- The command takes a location of an existing dataset as a positional argument, and optionally a path to where you want the dataset to be installed. If you do not specify a path, the dataset will be installed into the current directory, with the original name of the dataset.

- If a dataset is installed inside of a dataset as a subdataset, the --dataset/-d option needs to specify the root of the superdataset.

- The source can be a URL (for example of a GitHub repository, as in section *Install datasets* (page 46)), but also paths, or open data collections.

- After **datalad clone**, only small files and metadata about file availability are present locally. To retrieve actual file content of larger files, **datalad get PATH** downloads large file content on demand.

- **datalad status --annex** or **datalad status --annex all** are helpful to determine total repository size and the amount of data that is present locally.

- Remember: Super- and subdatasets have standalone histories. A superdataset only stores which version of the subdataset is currently used.

Now what I can do with that?

Simple, local workflows allow you to version control changing small files, for example your CV, your code, or a book that you are working on, but you can also add very large files to your datasets history. Currently, this can be considered "best-practice building": Frequent **datalad status** commands, **datalad save** commands to save dataset modifications, and concise COMMIT MESSAGEs are the main take aways from this. You can already explore the history of a dataset and you know about many types of provenance information captured by DataLad, but for now, its been only informative, and has not been used for anything more fancy. Later on, we will look into utilizing the history in order to undo mistakes, how the origin of files or datasets becomes

helpful when sharing datasets or removing file contents, and how to make changes to large content (as opposed to small content we have been modifying so far).

Additionally, you learned the basics on extending the `DataLad-101` dataset and consuming existing datasets: You have procedurally experienced how to install a dataset, and simultaneously you have learned a lot about the principles and features of DataLad datasets. Cloning datasets and getting their content allows you to consume published datasets. By nesting datasets within each other, you can re-use datasets in a modular fashion. While this may appear abstract, upcoming sections will demonstrate many examples of why this can be handy.

DATALAD, RUN!

7.1 Keeping track

In previous examples, with the exception of `datalad download-url`, all changes that happened to the dataset or the files it contains were saved to the dataset's history by hand. We added larger and smaller files and saved them, and we also modified smaller file contents and saved these modifications.

Often, however, files get changed by shell commands or by scripts. Consider a data scientist[74]. She has data files with numeric data, and code scripts in Python, R, Matlab or any other programming language that will use the data to compute results or figures. Such output is stored in new files, or modifies existing files.

But only a few weeks after these scripts were executed she finds it hard to remember which script was modified for which reason or created which output. How did this result came to be? Which script would she need to run again on which data to produce this particular figure?

In this section we will experience how DataLad can help to record the changes in a dataset after executing a script from the shell. Just as `datalad download-url` was able to associate a

---

[74] https://xkcd.com/1838/

file with its origin and store this information, we want to be able to associate a particular file with the commands, scripts, and inputs it was produced from, and thus capture and store full PROVENANCE.

Let's say, for example, that you enjoyed the longnow podcasts a lot, and you start a podcast-night with friends to wind down from all of the exciting DataLad lectures. They propose to make a list of speakers and titles to cross out what they've already listened to, and ask you to prepare such a list.

"Mhh... probably there is a DataLad way to do this... wasn't there also a note about metadata extraction at some point?" But as we're not that far into the lectures, you decide to write a short shell script to generate a text file that lists speaker and title name instead.

To do this, we're following a best practice that will reappear in the later section on YODA principles: Collecting all additional scripts that work with content of a subdataset *outside* of this subdataset, in a dedicated code/ directory, and collating the output of the execution of these scripts *outside* of the subdataset as well – and therefore not modifying the subdataset.

The motivation behind this will become clear in later sections, but for now we'll start with best-practice building. Therefore, create a subdirectory code/ in the DataLad-101 superdataset:

```
$ mkdir code
$ tree -d
.
├── books
├── code
└── recordings
    └── longnow
        ├── Long_Now__Conversations_at_The_Interval
        └── Long_Now__Seminars_About_Long_term_Thinking

6 directories
```

Inside of DataLad-101/code, create a simple shell script list_titles.sh. This script will carry out a simple task: It will loop through the file names of the .mp3 files and write out speaker names and talk titles in a very basic fashion. The content of this script is written below – the cat command will write it into the script.

W7.1 Here's a script for Windows users

Please use an editor of your choice to create a file list_titles.sh inside of the code directory. These should be the contents:

```
for i in recordings/longnow/Long_Now__Seminars*/*.mp3; do
    # get the filename
    base=$(basename "$i");
    # strip the extension
    base=${base%.mp3};
    # date as yyyy-mm-dd
    printf "${base%%__*}\t" | tr '_' '-';
    # name and title without underscores
    printf "${base#*__}\n" | tr '_' ' ';
done
```

Note that this is not identical to the one below – it lacks a few \ characters, which is a meaningful difference.

> **W7.2 Be mindful of hidden extensions when creating files!**
>
> By default, Windows does not show common file extensions when you view directory contents with a file explorer. Instead, it only displays the base of the file name and indicates the file type with the display icon. You can see if this is the case for you, too, by opening the books\ directory in a file explorer, and checking if the file extension (.pdf) is a part of the file name displayed underneath its PDF icon.
>
> Hidden file extensions can be a confusing source of errors, because some Windows editors (for example Notepad) automatically add a .txt extension to your files – when you save the script above under the name list_titles.sh, your editor may add an extension (list_titles.sh.txt), and the file explorer displays your file as list_titles.sh (hiding the .txt extension).
>
> To prevent confusion, configure the file explorer to always show you the file extension. For this, open the Explorer, click on the "View" tab, and tick the box "File name extensions".
>
> Beyond this, double check the correct naming of your file, ideally in the terminal.

```
$ cat << EOT > code/list_titles.sh
for i in recordings/longnow/Long_Now__Seminars*/*.mp3; do
    # get the filename
    base=\$(basename "\$i");
    # strip the extension
    base=\${base%.mp3};
    # date as yyyy-mm-dd
    printf "\${base%%__*}\t" | tr '_' '-';
    # name and title without underscores
    printf "\${base#*__}\n" | tr '_' ' ';
done
EOT
```

Save this script to the dataset.

```
$ datalad status
untracked: code (directory)
```

```
$ datalad save -m "Add short script to write a list of podcast speakers and titles"
add(ok): code/list_titles.sh (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Once we run this script, it will simply print dates, names and titles to your terminal. We can save its outputs to a new file recordings/podcasts.tsv in the superdataset by redirecting these outputs with bash code/list_titles.sh > recordings/podcasts.tsv.

Obviously, we could create this file, and subsequently save it to the superdataset. However, just as in the example about the data scientist, in a bit of time, we will forget how this file came into existence, or that the script code/list_titles.sh is associated with this file, and can be used to update it later on.

The **datalad run** command (datalad-run manual) can help with this. Put simply, it records a command's impact on a dataset. Put more technically, it will record a shell command, and **save** all changes this command triggered in the dataset – be that new files or changes to existing

files.

Let's try the simplest way to use this command: **datalad run**, followed by a commit message
(-m "a concise summary"), and the command that executes the script from the shell: bash
code/list_titles.sh > recordings/podcasts.tsv. It is helpful to enclose the command in
quotation marks.

Note that we execute the command from the root of the superdataset. It is recommended to
use **datalad run** in the root of the dataset you want to record the changes in, so make sure to
run this command from the root of DataLad-101.

```
$ datalad run -m "create a list of podcast titles" \
  "bash code/list_titles.sh > recordings/podcasts.tsv"
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/podcasts.tsv (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (notneeded: 1, ok: 1)
```

> **M7.1 Why is there a "notneeded" in the command summary?**
>
> If you have stumbled across the command execution summary save (notneeded: 1,
> ok: 1) and wondered what is "notneeded": the **datalad save** at the end of a **datalad
> run** will query all potential subdatasets *recursively* for modifications, and as there are no
> modifications in the longnow subdataset, this part of save returns a "notneeded" sum-
> mary. Thus, after a **datalad run**, you'll get a "notneeded" for every subdataset with no
> modifications in the execution summary.

Let's take a look into the history:

```
$ git log -p -n 1   # On Windows, you may just want to type "git log".
commit f0aab98d14a86fbe1100d991fe6fc46ef95e39c6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:34 2021 +0200

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "8e04afb0-af85-4070-be29-858d30d85017",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [],
     "outputs": [],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^

diff --git a/recordings/podcasts.tsv b/recordings/podcasts.tsv
new file mode 100644
index 0000000..f691b53
--- /dev/null
```

(continues on next page)

```
+++ b/recordings/podcasts.tsv
@@ -0,0 +1,206 @@
+2003-11-15       Brian Eno  The Long Now
+2003-12-13       Peter Schwartz  The Art Of The Really Long View
+2004-01-10       George Dyson  There s Plenty of Room at the Top  Long term Thinking␣
↪About Large scale Computing
+2004-02-14       James Dewar  Long term Policy Analysis
```

The commit message we have supplied with -m directly after **datalad run** appears in our history as a short summary. Additionally, the output of the command, recordings/podcasts.tsv, was saved right away.

But there is more in this log entry, a section in between the markers

=== Do not change lines below === and

^^^ Do not change lines above ^^^.

This is the so-called run record – a recording of all of the information in the **datalad run** command, generated by DataLad. In this case, it is a very simple summary. One informative part is highlighted: "cmd": "bash code/list_titles.sh" is the command that was run in the terminal. This information therefore maps the command, and with it the script, to the output file, in one commit. Nice, isn't it?

Arguably, the RUN RECORD is not the most human-readable way to display information. This representation however is less for the human user (the human user should rely on their informative commit message), but for DataLad, in particular for the **datalad rerun** command, which you will see in action shortly. This run record is machine-readable provenance that associates an output with the command that produced it.

You have probably already guessed that every **datalad run** command ends with a datalad save. A logical consequence from this fact is that any **datalad run** does not result in any changes in a dataset (no modification of existing content; no additional files) will not produce any record in the dataset's history (just as a **datalad save** with no modifications present will not create a history entry). Try to run the exact same command as before, and check whether anything in your log changes:

```
$ datalad run -m "Try again to create a list of podcast titles" \
  "bash code/list_titles.sh > recordings/podcasts.tsv"
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
action summary:
  save (notneeded: 2)
```

```
$ git log --oneline
f0aab98 [DATALAD RUNCMD] create a list of podcast titles
a76bf3b Add short script to write a list of podcast speakers and titles
f5fcfd5 Add note on datalad clone
6cd43dd [DATALAD] modified subdataset properties
```

The most recent commit is still the **datalad run** command from before, and there was no second **datalad run** commit created.

The **datalad run** can therefore help you to keep track of what you are doing in a dataset and capture provenance of your files: When, by whom, and how exactly was a particular file created or modified? The next sections will demonstrate how to make use of this information, and also

how to extend the command with additional arguments that will prove to be helpful over the course of this chapter.

7.2 DataLad, Re-Run!

So far, you created a `.tsv` file of all speakers and talk titles in the `longnow/` podcasts subdataset. Let's actually take a look into this file now:

```
$ less recordings/podcasts.tsv
2003-11-15      Brian Eno  The Long Now
2003-12-13      Peter Schwartz  The Art Of The Really Long View
2004-01-10      George Dyson  There s Plenty of Room at the Top  Long term Thinking␣
↪About Large scale Computing
2004-02-14      James Dewar  Long term Policy Analysis
2004-03-13      Rusty Schweickart  The Asteroid Threat Over the Next 100 000 Years
2004-04-10      Daniel Janzen  Third World Conservation  It s ALL Gardening
2004-05-15      David Rumsey  Mapping Time
2004-06-12      Bruce Sterling  The Singularity  Your Future as a Black Hole
2004-07-10      Jill Tarter  The Search for Extra terrestrial Intelligence  Necessarily␣
↪a Long term Strategy
2004-08-14      Phillip Longman  The Depopulation Problem
2004-09-11      Danny Hillis  Progress on the 10 000 year Clock
2004-10-16      Paul Hawken  The Long Green
2004-11-13      Michael West  The Prospects of Human Life Extension
2004-12-04      Ken Dychtwald  The Consequences of Human Life Extension
```

Not too bad, and certainly good enough for the podcast night people. What's been cool about creating this file is that it was created with a script within a **datalad run** command. Thanks to **datalad run**, the output file `podcasts.tsv` is associated with the script it generated.

Upon reviewing the list you realized that you made a mistake, though: you only listed the talks in the SALT series (the `Long_Now__Seminars_About_Long_term_Thinking/` directory), but not in the `Long_Now__Conversations_at_The_Interval/` directory. Let's fix this in the script. Replace the contents in code/`list_titles.sh` with the following, fixed script:

> **W7.3 Here's a script adjustment for Windows users**
>
> Please use an editor of your choice to replace the contents of `list_titles.sh` inside of the code directory with the following:
>
> ```
> for i in recordings/longnow/Long_Now*/*.mp3; do
>     # get the filename
>     base=$(basename "$i");
>     # strip the extension
>     base=${base%.mp3};
>     # date as yyyy-mm-dd
>     printf "${base%%__*}\t" | tr '_' '-';
>     # name and title without underscores
>     printf "${base#*__}\n" | tr '_' ' ';
> done
> ```

```
$ cat << EOT >| code/list_titles.sh
for i in recordings/longnow/Long_Now*/*.mp3; do
```

(continues on next page)

```
  # get the filename
  base=\$(basename "\$i");
  # strip the extension
  base=\${base%.mp3};
  printf "\${base%%__*}\t" | tr '_' '-';
  # name and title without underscores
  printf "\${base#*__}\n" | tr '_' ' ';

done
EOT
```

Because the script is now modified, save the modifications to the dataset. We can use the shorthand "BF" to denote "Bug fix" in the commit message.

```
$ datalad status
 modified: code/list_titles.sh (file)
```

```
$ datalad save -m "BF: list both directories content" \
  code/list_titles.sh
add(ok): code/list_titles.sh (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

What we *could* do is run the same **datalad run** command as before to recreate the file, but now with all of the contents:

```
# do not execute this!
$ datalad run -m "create a list of podcast titles" \
  "bash code/list_titles.sh > recordings/podcasts.tsv"
```

However, think about any situation where the command would be longer than this, or that is many months past the first execution. It would not be easy to remember the command, nor would it be very convenient to copy it from the run record.

Luckily, a fellow student remembered the DataLad way of re-executing a run command, and he's eager to show it to you.

"In order to re-execute a **datalad run** command, find the commit and use its SHASUM (or a TAG, or anything else that Git understands) as an argument for the **datalad rerun** command (datalad-rerun manual)! That's it!", he says happily.

So you go ahead and find the commit SHASUM in your history:

```
$ git log -n 2
commit 85330eaca98a1e1a6a6fa0a5d8118c1cf08cb9eb
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:37 2021 +0200

    BF: list both directories content

commit f0aab98d14a86fbe1100d991fe6fc46ef95e39c6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:34 2021 +0200
```

```
    [DATALAD RUNCMD] create a list of podcast titles
```

Take that shasum and paste it after **datalad rerun** (the first 6-8 characters of the shasum would be sufficient, here we're using all of them).

```
$ datalad rerun f0aab98d14a86fbe1100d991fe6fc46ef95e39c6
[INFO] run commit f0aab98; (create a list of ...)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/podcasts.tsv (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (notneeded: 1, ok: 1)
  unlock (notneeded: 1)
```

Now DataLad has made use of the run record, and re-executed the original command based on the information in it. Because we updated the script, the output podcasts.tsv has changed and now contains the podcast titles of both subdirectories. You've probably already guessed it, but the easiest way to check whether a **datalad rerun** has changed the desired output file is to check whether the rerun command appears in the datasets history: If a **datalad rerun** does not add or change any content in the dataset, it will also not be recorded in the history.

```
$ git log -n 1
commit d5edb1866bcbafc998233e819432321185e51e85
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:38 2021 +0200

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [
      "f0aab98d14a86fbe1100d991fe6fc46ef95e39c6"
     ],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "8e04afb0-af85-4070-be29-858d30d85017",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [],
     "outputs": [],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^
```

In the dataset's history, we can see that a new **datalad run** was recorded. This action is committed by DataLad under the original commit message of the run command, and looks just like the previous **datalad run** commit apart from the execution time.

Two cool tools that go beyond the **git log** are the **datalad diff** (datalad-diff manual) and **git diff** commands. Both commands can report differences between two states of a dataset. Thus, you can get an overview of what changed between two commits. Both commands have a similar, but not identical structure: **datalad diff** compares one state (a commit specified with -f/--from, by default the latest change) and another state from the dataset's history (a commit

specified with -t/--to). Let's do a **datalad diff** between the current state of the dataset and the previous commit (called "HEAD~1" in Git terminology[75]):

> **W7.4 please use datalad diff –from master –to HEAD 1**
>
> While this example works on Unix file systems, it will not provide the same output on Windows. This is due to different file handling on Windows. When executing this command, you will see *all* files being modified between the most recent and the second-most recent commit. On a technical level, this is correct given the underlying file handling on Windows, and chapter *Under the hood: git-annex* (page 83) will shed light on why that is.
>
> For now, to get the same output as shown in the code snippet below, use the following command where main (or master) is the name of your default branch:
>
> ```
> datalad diff --from main --to HEAD~1
> ```
>
> The --from argument specifies a different starting point for the comparison - the main or MASTER BRANCH, which would be the starting point on most Unix-based systems.

```
$ datalad diff --to HEAD~1
 modified: recordings/podcasts.tsv (file)
```

This indeed shows the output file as "modified". However, we do not know what exactly changed. This is a task for **git diff** (get out of the diff view by pressing q):

```
$ git diff HEAD~1
diff --git a/recordings/podcasts.tsv b/recordings/podcasts.tsv
index f691b53..d77891d 100644
--- a/recordings/podcasts.tsv
+++ b/recordings/podcasts.tsv
@@ -1,3 +1,31 @@
+2017-06-09      How Digital Memory Is Shaping Our Future  Abby Smith Rumsey
+2017-06-09      Pace Layers Thinking  Stewart Brand  Paul Saffo
+2017-06-09      Proof  The Science of Booze  Adam Rogers
+2017-06-09      Seveneves at The Interval  Neal Stephenson
+2017-06-09      Talking with Robots about Architecture  Jeffrey McGrew
+2017-06-09      The Red Planet for Real  Andy Weir
+2017-07-03      Transforming Perception  One Sense at a Time  Kara Platoni
+2017-08-01      How Climate Will Evolve Government and Society  Kim Stanley Robinson
+2017-09-01      Envisioning Deep Time  Jonathon Keats
+2017-10-01      Thinking Long term About the Evolving Global Challenge  The Refugee␣
→Reality
+2017-11-01      The Web In An Eye Blink  Jason Scott
+2017-12-01      Ideology in our Genes  The Biological Basis for Political Traits  Rose␣
→McDermott
+2017-12-07      Can Democracy Survive the Internet   Nathaniel Persily
+2018-01-02      The New Deal You Don t Know  Louis Hyman
```

This output actually shows the precise changes between the contents created with the first version of the script and the second script with the bug fix. All of the files that are added after the second directory was queried as well are shown in the diff, preceded by a +.

Quickly create a note about these two helpful commands in notes.txt:

---

[75] The section *Back and forth in time* (page 245) will elaborate more on common GIT commands and terminology.

```
$ cat << EOT >> notes.txt
There are two useful functions to display changes between two
states of a dataset: "datalad diff -f/--from COMMIT -t/--to COMMIT"
and "git diff COMMIT COMMIT", where COMMIT is a shasum of a commit
in the history.

EOT
```

Finally, save this note.

```
$ datalad save -m "add note datalad and git diff"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Note that **datalad rerun** can re-execute the run records of both a **datalad run** or a **datalad rerun** command, but not with any other type of datalad command in your history such as a **datalad save** on results or outputs after you executed a script. Therefore, make it a habit to record the execution of scripts by plugging it into **datalad run**.

This very basic example of a **datalad run** is as simple as it can get, but it is already convenient from a memory-load perspective: Now you do not need to remember the commands or scripts involved in creating an output. DataLad kept track of what you did, and you can instruct it to "rerun" it. Also, incidentally, we have generated PROVENANCE information. It is now recorded in the history of the dataset how the output podcasts.tsv came into existence. And we can interact with and use this provenance information with other tools than from the machine-readable run record. For example, to find out who (or what) created or modified a file, give the file path to **git log** (prefixed by --):

> ### ⊞ W7.5 use "git log master – recordings/podcasts.tsv"
>
> A previous Windows Wit already advised to append main or master, the common "default BRANCH", to any command that starts with git log. Here, the last part of the command specifies a file (-- recordings/podcasts.tsv). Please append main or master to git log, prior to the file specification.

```
$ git log -- recordings/podcasts.tsv
commit d5edb1866bcbafc998233e819432321185e51e85
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:38 2021 +0200

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [
      "f0aab98d14a86fbe1100d991fe6fc46ef95e39c6"
     ],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "8e04afb0-af85-4070-be29-858d30d85017",
     "exit": 0,
     "extra_inputs": [],
```

<div align="right">(continues on next page)</div>

```
    "inputs": [],
    "outputs": [],
    "pwd": "."
   }
   ^^^ Do not change lines above ^^^

commit f0aab98d14a86fbe1100d991fe6fc46ef95e39c6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:34 2021 +0200

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "8e04afb0-af85-4070-be29-858d30d85017",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [],
     "outputs": [],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^
```

Neat, isn't it?

Still, this **datalad run** was very simple. The next section will demonstrate how **datalad run** becomes handy in more complex standard use cases: situations with *locked* contents.

But prior to that, make a note about **datalad run** and **datalad rerun** in your notes.txt file.

```
$ cat << EOT >> notes.txt
The datalad run command can record the impact a script or command has
on a Dataset. In its simplest form, datalad run only takes a commit
message and the command that should be executed.

Any datalad run command can be re-executed by using its commit shasum
as an argument in datalad rerun CHECKSUM. DataLad will take
information from the run record of the original commit, and re-execute
it. If no changes happen with a rerun, the command will not be written
to history. Note: you can also rerun a datalad rerun command!

EOT
```

Finally, save this note.

```
$ datalad save -m "add note on basic datalad run and datalad rerun"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

7.3 Input and output

In the previous two sections, you created a simple `.tsv` file of all speakers and talk titles in the `longnow/` podcasts subdataset, and you have re-executed a **datalad run** command after a bug-fix in your script.

But these previous **datalad run** and **datalad rerun** command were very simple. Maybe you noticed some values in the `run record` were empty: `inputs` and `outputs` for example did not have an entry. Let's experience a few situations in which these two arguments can become necessary.

In our DataLad-101 course we were given a group assignment. Everyone should give a small presentation about an open DataLad dataset they found. Conveniently, you decided to settle for the longnow podcasts right away. After all, you know the dataset quite well already, and after listening to almost a third of the podcasts and enjoying them a lot, you also want to recommend them to the others.

Almost all of the slides are ready, but what's still missing is the logo of the longnow podcasts. Good thing that this is part of the subdataset, so you can simply retrieve it from there.

The logos (one for the SALT series, one for the Interval series – the two directories in the subdataset) were originally extracted from the podcasts metadata information by DataLad. In a while, we will dive into the metadata aggregation capabilities of DataLad, but for now, let's just use the logos instead of finding out where they come from – this will come later. As part of the metadata of the dataset, the logos are in the hidden paths `.datalad/feed_metadata/logo_salt.jpg` and `.datalad/feed_metadata/logo_interval.jpg`:

```
$ ls recordings/longnow/.datalad/feed_metadata/*jpg
recordings/longnow/.datalad/feed_metadata/logo_interval.jpg
recordings/longnow/.datalad/feed_metadata/logo_salt.jpg
```

For the slides you decide to prepare images of size 400x400 px, but the logos' original size is much larger (both are 3000x3000 pixel). Therefore let's try to resize the images – currently, they're far too large to fit on a slide.

To resize an image from the command line we can use the Unix command `convert -resize` from the [ImageMagick tool][76]. The command takes a new size in pixels as an argument, a path to the file that should be resized, and a filename and path under which a new, resized image will be saved. To resize one image to 400x400 px, the command would thus be `convert -resize 400x400 path/to/file.jpg path/to/newfilename.jpg`.

> **W7.6 Tool installation**
>
> [ImageMagick][77] is not installed on Windows systems by default. To use it, you need to install it, using the provided [Windows Binary Release on the Download page][78].
> During installation, it is important to install the tool into a place where it is easily accessible to your terminal. The easiest way to do this is by selecting the installation directory to be within `Miniconda3` (i.e., just as during the installation of git-annex as described in *Installation and configuration* (page 10), install is into `C:\Users\<user-name>\miniconda3\Library`). Do also make sure to tick the box "install legacy commands" in the installation wizard.

---

[76] https://imagemagick.org/index.php

⁷⁷ https://imagemagick.org/index.php
⁷⁸ https://imagemagick.org/script/download.php

Remembering the last lecture on **datalad run**, you decide to plug this into **datalad run**. Even though this is not a script, it is a command, and you can wrap commands like this conveniently with **datalad run**. Because they will be quite long, we line break the commands in the upcoming examples for better readability – in your terminal, you can always write the commands into a single line.

```
$ datalad run -m "Resize logo for slides" \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg␣
↪recordings/salt_logo_small.jpg"
[INFO] == Command start (output follows) =====
convert-im6.q16: unable to open image `recordings/longnow/.datalad/feed_metadata/logo_
↪salt.jpg': No such file or directory @ error/blob.c/OpenBlob/2924.
convert-im6.q16: no images defined `recordings/salt_logo_small.jpg' @ error/convert.c/
↪ConvertImageCommand/3229.
[INFO] == Command exit (modification check follows) =====
[INFO] The command had a non-zero exit code. If this is expected, you can save the␣
↪changes with 'datalad save -d . -r -F .git/COMMIT_EDITMSG'
CommandError: 'convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_
↪salt.jpg recordings/salt_logo_small.jpg' failed with exitcode 1 under /home/me/dl-101/
↪DataLad-101
```

Oh, crap! Why didn't this work?

Let's take a look at the error message DataLad provides. In general, these error messages might seem wordy, and maybe a bit intimidating as well, but usually they provide helpful information to find out what is wrong. Whenever you encounter an error message, make sure to read it, even if it feels like a mushroom cloud exploded in your terminal.

A **datalad run** error message has several parts. The first starts after

```
[INFO ] == Command start (output follows) =====.
```

This is displaying errors that the terminal command threw: The convert tool complains that it can not open the file, because there is "No such file or directory".

The second part starts after

```
[INFO ] == Command exit (modification check follows) =====.
```

DataLad adds information about a "non-zero exit code". A non-zero exit code indicates that something went wrong[81]. In principle, you could go ahead and google what this specific exit status indicates. However, the solution might have already occurred to you when reading the first error report: The file is not present.

How can that be?

"Right!", you exclaim with a facepalm. Just as the .mp3 files, the .jpg file content is not present locally after a **datalad clone**, and we did not **datalad get** it yet!

This is where the -i/--input option for a datalad run becomes useful. The content of everything that is specified as an input will be retrieved prior to running the command.

---

[81] In shell programming, commands exit with a specific code that indicates whether they failed, and if so, how. Successful commands have the exit code zero. All failures have exit codes greater than zero. A few lines lower, DataLad even tells us the specific error code: The command failed with exit code 1.

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg␣
↪recordings/salt_logo_small.jpg"
# or shorter:
$ datalad run -m "Resize logo for slides" \
-i "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg␣
↪recordings/salt_logo_small.jpg"
[INFO] Making sure inputs are available (this may take some time)
get(ok): recordings/longnow/.datalad/feed_metadata/logo_salt.jpg (file) [from web...]
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/salt_logo_small.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 1, ok: 1)
  save (notneeded: 1, ok: 1)
```

Cool! You can see in this output that prior to the data command execution, DataLad did a **datalad get**. This is useful for several reasons. For one, it saved us the work of manually getting content. But moreover, this is useful for anyone with whom we might share the dataset: With an installed dataset one can very simply rerun **datalad run** commands if they have the input argument appropriately specified. It is therefore good practice to specify the inputs appropriately. Remember from section *Install datasets* (page 46) that **datalad get** will only retrieve content if it is not yet present, all input already downloaded will not be downloaded again – so specifying inputs even though they are already present will not do any harm.

> 💡 **M7.2 What if there are several inputs?**
>
> Often, a command needs several inputs. In principle, every input (which could be files, directories, or subdatasets) gets its own -i/--input flag. However, you can make use of GLOBBING. For example,
>
> ```
> datalad run --input "*.jpg" "COMMAND"
> ```
>
> will retrieve all .jpg files prior to command execution.

If outputs already exist…

> 🪟 **W7.7 Good news! Here is something that is easier on Windows**
>
> The section below describes something that is very confusing for people that have just started with DataLad: Some files in a dataset can't be modified, and if one tries, it results in a "permission denied" error. Why is that? The remainder of this section and the upcoming chapter *Under the hood: git-annex* (page 83) contain a procedural explanation. However: This doesn't happen on Windows. The "unlocking" that is necessary on almost all other systems to modify a file is already done on Windows. Thus, all files in your dataset will be readily modifiable, sparing you the need to adjust to the unexpected behavior that is described below. While it is easier, it isn't a "more useful" behavior, though. A different Windows Wit in the next chapter will highlight how it rather is a

> suboptimal workaround.
>
> Please don't skip the next section – it is useful to know how datasets behave on other systems. Just be mindful that you will not encounter the errors that the handbook displays next. And while this all sounds quite cryptic and vague, an upcoming Windows Wit will provide more information.

Looking at the resulting image, you wonder whether 400x400 might be a tiny bit to small. Maybe we should try to resize it to 450x450, and see whether that looks better?

Note that we can not use a **datalad rerun** for this: if we want to change the dimension option in the command, we have to define a new **datalad run** command.

To establish best-practices, let's specify the input even though it is already present:

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \
"convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg␣
→recordings/salt_logo_small.jpg"
# or shorter:
$ datalad run -m "Resize logo for slides" \
-i "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \
"convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg␣
→recordings/salt_logo_small.jpg"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
convert-im6.q16: unable to open image `recordings/salt_logo_small.jpg': Permission denied␣
→@ error/blob.c/OpenBlob/2924.
[INFO] == Command exit (modification check follows) =====
[INFO] The command had a non-zero exit code. If this is expected, you can save the␣
→changes with 'datalad save -d . -r -F .git/COMMIT_EDITMSG'
CommandError: 'convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_
→salt.jpg recordings/salt_logo_small.jpg' failed with exitcode 1 under /home/me/dl-101/
→DataLad-101
```

Oh wtf... *What is it now?*

A quick glimpse into the error message shows a different error than before: The tool complains that it is "unable to open" the image, because the "Permission [is] denied".

We have not seen anything like this before, and we need to turn to our lecturer for help. Confused about what we might have done wrong, we raise our hand to ask the instructor. Knowingly, she smiles, and tells you about how DataLad protects content given to it:

"Content in your DataLad dataset is protected by GIT-ANNEX from accidental changes" our instructor begins.

"Wait!" we interrupt. "First off, that wasn't accidental. And second, I was told this course does not have git-annex-101 as a prerequisite?"

"Yes, hear me out" she says. "I promise you two different solutions at the end of this explanation, and the concept behind this is quite relevant".

DataLad usually gives content to GIT-ANNEX to store and track. git-annex, let's just say, takes this task *really* seriously. One of its features that you have just experienced is that it *locks* content.

If files are *locked down*, their content can not be modified. In principle, that's not a bad thing: It could be your late grandma's secret cherry-pie recipe, and you do not want to *accidentally*

change that. Therefore, a file needs to be consciously *unlocked* to apply modifications.

In the attempt to resize the image to 450x450 you tried to overwrite `recordings/salt_logo_small.jpg`, a file that was given to DataLad and thus protected by git-annex.

There is a DataLad command that takes care of unlocking file content, and thus making locked files modifiable again: **datalad unlock** (datalad-unlock manual). Let us check out what it does:

> **W7.8 What happens if I run this on Windows?**
>
> Nothing. All of the files in your dataset are always unlocked, and actually *can* not be locked at all. Consequently, there will be nothing to show for `datalad status` afterwards (as shown a few paragraphs below). This is due to a file system limitation, and will be explained in more detail in chapter *Under the hood: git-annex* (page 83).

```
$ datalad unlock recordings/salt_logo_small.jpg
unlock(ok): recordings/salt_logo_small.jpg (file)
```

Well, unlock(ok) does not sound too bad for a start. As always, we feel the urge to run a **datalad status** on this:

```
$ datalad status
 modified: recordings/salt_logo_small.jpg (file)
```

"Ah, do not mind that for now", our instructor says, and with a wink she continues: "We'll talk about symlinks and object trees a while later". You are not really sure whether that's a good thing, but you have a task to focus on. Hastily, you run the command right from the terminal:

```
$ convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg↵
→recordings/salt_logo_small.jpg
```

Hey, no permission denied error! You note that the instructor still stands right next to you. "Sooo... now what do I do to *lock* the file again?" you ask.

"Well... what you just did there was quite suboptimal. Didn't you want to use **datalad run**? But, anyway, in order to lock the file again, you would need to run a **datalad save**."

```
$ datalad save -m "resized picture by hand"
add(ok): recordings/salt_logo_small.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

"So", you wonder aloud, "whenever I want to modify I need to **datalad unlock** it, do the modifications, and then **datalad save** it?"

"Well, this is certainly one way of doing it, and a completely valid workflow if you would do that outside of a **datalad run** command. But within **datalad run** there is actually a much easier way of doing this. Let's use the `--output` argument."

datalad run *retrieves* everything that is specified as `--input` prior to command execution, and it *unlocks* everything specified as `--output` prior to command execution. Therefore, whenever the output of a **datalad run** command already exists and is tracked, it should be specified as an argument in the `-o`/`--output` option.

> 💡 **M7.3 But what if I have a lot of outputs?**
>
> The use case here is simplistic – a single file gets modified. But there are commands and tools that create full directories with many files as an output, for example FSL[79], a neuro-imaging tool. The easiest way to specify this type of output is by supplying the directory name, or the directory name and a GLOBBING character, such as `-o directory/*.dat`. This would unlock all files with a `.dat` extension inside of `directory`. To glob for files in multiple levels of directories, set one `*` per level, for example `-o directory/*/*.dat` when operating with DataLad version <=`0.15.4`. If your version of DataLad is more recent than that, use `**` (a so-called globstar[80]) for a recursive glob through any number directories. And, just as for `-i/--input`, you could use multiple `--output` specifications.
>
> ___
> [79] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki
> [80] https://www.linuxjournal.com/content/globstar-new-bash-globbing-option

In order to execute **datalad run** with both the `-i/--input` and `-o/--output` flag and see their magic, let's crop the second logo, `logo_interval.jpg`:

> 🪟 **W7.9 Wait, would I need to specify outputs, too?**
>
> Given that nothing in your dataset is locked, is there a *need* for you to bother with creating `--output` flags? Not for you personally, if you only stay on your Windows machine. However, you will be doing others that you share your dataset with a favour if they are not using Windows – should you or others want to rerun a run record, `--output` flags will make it work on all operating systems.

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
--output "recordings/interval_logo_small.jpg" \
"convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg␣
↪recordings/interval_logo_small.jpg"

# or shorter:
$ datalad run -m "Resize logo for slides" \
-i "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
-o "recordings/interval_logo_small.jpg" \
"convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg␣
↪recordings/interval_logo_small.jpg"
[INFO] Making sure inputs are available (this may take some time)
get(ok): recordings/longnow/.datalad/feed_metadata/logo_interval.jpg (file) [from web...]
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/interval_logo_small.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 1, ok: 1)
  save (notneeded: 1, ok: 1)
```

This time, with both `--input` and `--output` options specified, DataLad informs about the **datalad get** operations it performs prior to the command execution, and **datalad run** executes the command successfully. It does *not* inform about any **datalad unlock** operation, because the output `recordings/interval_logo_small.jpg` does not exist before the command is run. Should you rerun this command however, the summary will include a statement about content

unlocking. You will see an example of this in the next section.

Note now how many individual commands a `datalad run` saves us: `datalad get`, `datalad unlock`, and `datalad save`! But even better: Beyond saving time *now*, running commands reproducibly and recorded with `datalad run` saves us plenty of time in the future as soon as we want to rerun a command, or find out how a file came into existence.

With this last code snippet, you have experienced a full `datalad run` command: commit message, input and output definitions (the order in which you give those two options is irrelevant), and the command to be executed. Whenever a command takes input or produces output you should specify this with the appropriate option.

Make a note of this behavior in your `notes.txt` file.

```
$ cat << EOT >> notes.txt
You should specify all files that a command takes as input with an
-i/--input flag. These files will be retrieved prior to the command
execution. Any content that is modified or produced by the command
should be specified with an -o/--output flag. Upon a run or rerun of
the command, the contents of these files will get unlocked so that
they can be modified.

EOT
```

Save yourself the preparation time

> ⚠️ **version requirement for --assume-ready**
>
> The option `--assume-ready` was introduced with DataLad version 0.14.1.

Its generally good practice to specify `--input` and `--output` even if your input files are already retrieved and your output files unlocked – it makes sure that a recomputation can succeed, even if inputs are not yet retrieved, or if output needs to be unlocked. However, the internal preparation steps of checking that inputs exist or that outputs are unlocked can take a bit of time, especially if it involves checking a large number of files.

Starting with **datalad version 0.14.1**, you can make use of the `--assume-ready` argument of `datalad run` if you want to avoid the expense of unnecessary preparation steps. Depending on whether your inputs are already retrieved, your outputs already unlocked (or not needed to be unlocked), or both, specify `--assume-ready` with the argument `inputs`, `outputs` or `both` and save yourself a few seconds, without sacrificing the ability to rerun your command under conditions in which the preparation would be necessary.

Placeholders

Just after writing the note, you had to relax your fingers a bit. "Man, this was so much typing. Not only did I need to specify the inputs and outputs, I also had to repeat all of these lengthy paths in the command line call..." you think.

There is a neat little trick to spare you half of this typing effort, though: *Placeholders* for inputs and outputs. This is how it works:

Instead of running

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
--output "recordings/interval_logo_small.jpg" \
"convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg␣
↪recordings/interval_logo_small.jpg"
```

you could shorten this to

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
--output "recordings/interval_logo_small.jpg" \
"convert -resize 450x450 {inputs} {outputs}"
```

The placeholder {inputs} will expand to the path given as --input, and the placeholder {outputs} will expand to the path given as --output. This means instead of writing the full paths in the command, you can simply reuse the --input and --output specification done before.

> **M7.4 What if I have multiple inputs or outputs?**
>
> If multiple values are specified, e.g., as in
>
> ```
> $ datalad run -m "move a few files around" \
> --input "file1" --input "file2" --input "file3" \
> --output "directory_a/" \
> "mv {inputs} {outputs}"
> ```
>
> the values will be joined by a space like this:
>
> ```
> $ datalad run -m "move a few files around" \
> --input "file1" --input "file2" --input "file3" \
> --output "directory_a/" \
> "mv file1 file2 file3 directory_a/"
> ```
>
> The order of the values will match that order from the command line.
> If you use globs for input specification, as in
>
> ```
> $ datalad run -m "move a few files around" \
> --input "file*" \
> --output "directory_a/" \
> "mv {inputs} {outputs}"
> ```
>
> the globs will expanded in alphabetical order (like bash):
>
> ```
> $ datalad run -m "move a few files around" \
> --input "file1" --input "file2" --input "file3" \
> --output "directory_a/" \
> "mv file1 file2 file3 directory_a/"
> ```
>
> If the command only needs a subset of the inputs or outputs, individual values can be accessed with an integer index, e.g., {inputs[0]} for the very first input.

> **M7.5 ... wait, what if I need a curly bracket in my datalad run call?**
>
> If your command call involves a { or } character, you will need to escape this brace character by doubling it, i.e., {{ or }}.

Dry-running your run call

datalad run commands can become confusing and long, especially when you make heavy use of placeholders or wrap a complex bash commands. To better anticipate what you will be running, or help debug a failed command, you can make use of the `--dry-run` flag of `datalad run`. This option needs a mode specification (`--dry-run=basic` or `dry-run=command`), followed by the `run` command you want to execute, and it will decipher the commands elements: The mode `command` will display the command that is about to be ran. The mode `basic` will report a few important details about the execution: Apart from displaying the command that will be ran, you will learn *where* the command runs, what its *inputs* are (helpful if your `--input` specification includes a GLOBBING term), and what its *outputs* are.

7.4 Clean desk

Just now you realize that you need to fit both logos onto the same slide. "Ah, damn, I might then really need to have them 400 by 400 pixel to fit", you think. "Good that I know how to not run into the permission denied errors anymore!"

Therefore, we need to do the **datalad run** command yet again - we wanted to have the image in 400x400 px size. "Now this definitely will be the last time I'm running this", you think.

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
--output "recordings/interval_logo_small.jpg" \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg␣
↪recordings/interval_logo_small.jpg"
run(impossible): /home/me/dl-101/DataLad-101 (dataset) [clean dataset required to detect␣
↪changes from command; use `datalad status` to inspect unsaved changes]
```

Oh for f**** sake... run is "impossible"?

Weird. After the initial annoyance about yet another error message faded, and you read on, DataLad informs that a "clean dataset" is required. Run a **datalad status** to see what is meant by this:

```
$ datalad status
 modified: notes.txt (file)
```

Ah right. We forgot to save the notes we added, and thus there are unsaved modifications present in `DataLad-101`. But why is this a problem?

By default, at the end of a **datalad run** is a **datalad save**. Remember the section *Populate a dataset* (page 37): A general **datalad save** without a path specification will save *all* of the modified or untracked contents to the dataset.

Therefore, in order to not mix any changes in the dataset that are unrelated to the command plugged into **datalad run**, by default it will only run on a clean dataset with no changes or untracked files present.

There are two ways to get around this error message: The more obvious – and recommended – one is to save the modifications, and run the command in a clean dataset. We will try this way with the `logo_interval.jpg`. It would look like this: First, save the changes,

```
$ datalad save -m "add additional notes on run options"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

and then try again:

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
--output "recordings/interval_logo_small.jpg" \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg␣
↪recordings/interval_logo_small.jpg"
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): recordings/interval_logo_small.jpg (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/interval_logo_small.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 2)
  save (notneeded: 1, ok: 1)
  unlock (ok: 1)
```

Note how in this execution of **datalad run**, output unlocking was actually necessary and Data-Lad provides a summary of this action in its output.

Add a quick addition to your notes about this way of cleaning up prior to a **datalad run**:

```
$ cat << EOT >> notes.txt
Important! If the dataset is not "clean" (a datalad status output is
empty), datalad run will not work - you will have to save
modifications present in your dataset.
EOT
```

A way of executing a **datalad run** *despite* an "unclean" dataset, though, is to add the `--explicit` flag to **datalad run**. We will try this flag with the remaining logo_salt.jpg. Note that we have an "unclean dataset" again because of the additional note in notes.txt.

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \
--output "recordings/salt_logo_small.jpg" \
--explicit \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg␣
↪recordings/salt_logo_small.jpg"
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): recordings/salt_logo_small.jpg (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/salt_logo_small.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 2)
```

```
  save (ok: 1)
  unlock (ok: 1)
```

With this flag, DataLad considers the specification of inputs and outputs to be "explicit". It does not warn if the repository is dirty, but importantly, it **only** saves modifications to the *listed outputs* (which is a problem in the vast amount of cases where one does not exactly know which outputs are produced).

> ⚠️ **Put explicit first!**
>
> The --explicit flag has to be given anywhere *prior* to the command that should be run – the command needs to be the last element of a **datalad run** call.

A **datalad status** will show that your previously modified notes.txt is still modified:

```
$ datalad status
 modified: notes.txt (file)
```

Add an additional note on the --explicit flag, and finally save your changes to notes.txt.

```
$ cat << EOT >> notes.txt
A suboptimal alternative is the --explicit flag, used to record only
those changes done to the files listed with --output flags.

EOT
```

```
$ datalad save -m "add note on clean datasets"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

To conclude this section on **datalad run**, take a look at the last **datalad run** commit to see a RUN RECORD with more content:

```
$ git log -p -n 2
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:21:52 2021 +0200

    [DATALAD RUNCMD] Resize logo for slides

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.
↪jpg recordings/salt_logo_small.jpg",
     "dsid": "8e04afb0-af85-4070-be29-858d30d85017",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [
      "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg"
     ],
     "outputs": [
```

```
    "recordings/salt_logo_small.jpg"
  ],
  "pwd": "."
}
^^^ Do not change lines above ^^^

diff --git a/recordings/salt_logo_small.jpg b/recordings/salt_logo_small.jpg
index b6a0a1d..55ada0f 120000
--- a/recordings/salt_logo_small.jpg
+++ b/recordings/salt_logo_small.jpg
@@ -1 +1 @@
```

7.5 Summary

In the last four sections, we demonstrated how to create a proper **datalad run** command, and discovered the concept of *locked* content.

- **datalad run** records and saves the changes a command makes in a dataset. That means that modifications to existing content or new content are associated with a specific command and saved to the dataset's history. Essentially, **datalad run** helps you to keep track of what you do in your dataset by capturing all PROVENANCE.

- A **datalad run** command generates a run record in the commit. This RUN RECORD can be used by datalad to re-execute a command with **datalad rerun SHASUM**, where SHASUM is the commit hash of the **datalad run** command that should be re-executed.

- If a **datalad run** or **datalad rerun** does not modify any content, it will not write a record to history.

- With any **datalad run**, specify a commit message, and whenever appropriate, specify its inputs to the executed command (using the -i/--input flag) and/or its output (using the -o/ --output flag). The full command structure is:

  ```
  $ datalad run -m "commit message here" --input "path/to/input/" --output "path/to/
  ↪output" "command"
  ```

- Anything specified as input will be retrieved if necessary with a **datalad get** prior to command execution. Anything specified as output will be unlocked prior to modifications.

- It is good practice to specify input and output to ensure that a **datalad rerun** works, and to capture the relevant elements of a computation in a machine-readable record. If you want to spare yourself preparation time in case everything is already retrieved and unlocked, you can use --assume-ready {input|output|both} to skip a check on whether inputs are already present or outputs already unlocked (requires DataLad version 0.14.1 or later).

- Getting and unlocking content is not only convenient for yourself, but enormously helpful for anyone you share your dataset with, but this will be demonstrated in an upcoming section in detail.

- To execute a **datalad run** or **datalad rerun**, a **datalad status** either needs to report that the dataset has no uncommitted changes (the dataset state should be "clean"), or the command needs to be extended with the --explicit option.
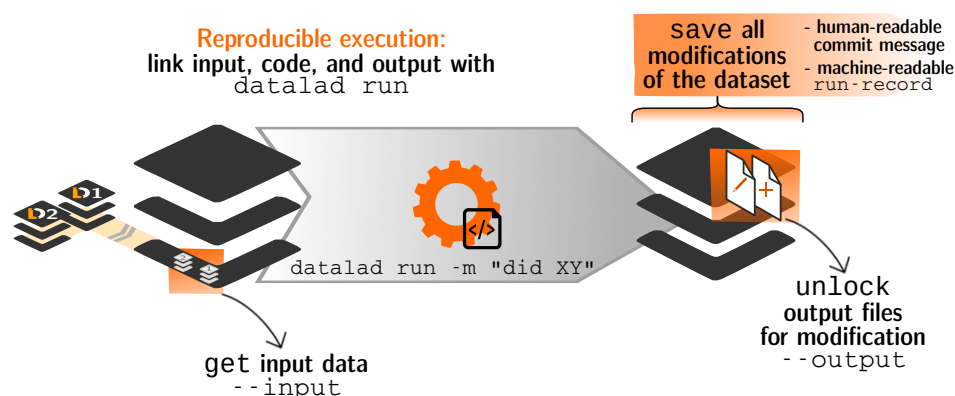
Fig. 7.1: Overview of `datalad run`.

Now what I can do with that?

You have procedurally experienced how to use **datalad run** and **datalad rerun**. Both of these commands make it easier for you and others to associate changes in a dataset with a script or command, and are helpful as the exact command for a given task is stored by DataLad, and does not need to be remembered.

Furthermore, by experiencing many common error messages in the context of **datalad run** commands, you have gotten some clues on where to look for problems, should you encounter those errors in your own work.

Lastly, we've started to unveil some principles of GIT-ANNEX that are relevant to understanding how certain commands work and why certain commands may fail. We have seen that git-annex locks large files' content to prevent accidental modifications, and how the `--output` flag in **datalad run** can save us an intermediate **datalad unlock** to unlock this content. The next section will elaborate on this a bit more.

UNDER THE HOOD: GIT-ANNEX

A closer look at how and why things work

8.1 Data safety

Later in the day, after seeing and solving so many DataLad error messages, you fall tired into your bed. Just as you are about to fall asleep, a thought crosses your mind:

"I now know that tracked content in a dataset is protected by GIT-ANNEX. Whenever tracked contents are saved, they get locked and should not be modifiable. But... what about the notes that I have been taking since the first day? Should I not need to unlock them before I can modify them? And also the script! I was able to modify this despite giving it to DataLad to track, with no permission denied errors whatsoever! How does that work?"

This night, though, your question stays unanswered and you fall into a restless sleep filled with bad dreams about "permission denied" errors. The next day you're the first student in your lecturer's office hours.

"Oh, you're really attentive. This is a great question!" our lecturer starts to explain.

Do you remember that we created the DataLad-101 dataset with a specific configuration template? It was the -c text2git option we provided in the beginning of *Create a dataset* (page 34). It is because of this configuration that we can modify notes.txt without unlocking its content first.

The second commit message in our datasets history summarizes this (outputs are shortened):

```
$ git log --reverse --oneline
9098f56 [DATALAD] new dataset
a19b634 Instruct annex to add text files to Git
7ad3b1b add books on Python and Unix to read later
ca8582d add reference book about git
1930699 add beginners guide on bash
cc213c2 Add notes on datalad create
13cc7f7 add note on datalad save
2472210 [DATALAD] Recorded changes
6cd43dd [DATALAD] modified subdataset properties
```

Instead of giving text files such as your notes or your script to git-annex, the dataset stores it in GIT. But what does it mean if files are in Git instead of git-annex?

Well, procedurally it means that everything that is stored in git-annex is content-locked, and everything that is stored in Git is not. You can modify content stored in Git straight away, without unlocking it first.
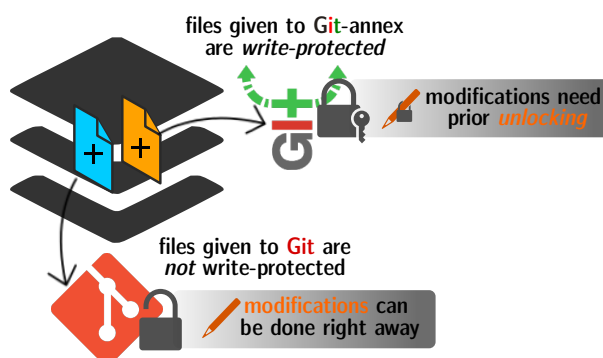
Fig. 8.1: A simplified overview of the tools that manage data in your dataset.

That's easy enough.

"So, first of all: If we hadn't provided the `-c text2git` argument, text files would get content-locked, too?". "Yes, indeed. However, there are also ways to later change how file content is handled based on its type or size. It can be specified in the `.gitattributes` file, using `annex.largefile` options. But there will be a lecture on that[82]."

"Okay, well, second: Isn't it much easier to just not bother with locking and unlocking, and have everything 'stored in Git'? Even if **datalad run** takes care of unlocking content, I do not see the point of git-annex", you continue.

---

[82] If you cannot wait to read about `.gitattributes` and other configuration files, jump ahead to chapter *Tuning datasets to your needs* (page 112), starting with section *DIY configurations* (page 112).

Here it gets tricky. To begin with the most important, and most straight-forward fact: It is not possible to store large files in Git. This is because Git would very quickly run into severe performance issues. For this reason, GITHUB, a well-known hosting site for projects using Git, for example does not allow files larger than a few dozen MB of size.

For now, we have solved the mystery of why text files can be modified without unlocking, and this is a small improvement in the vast amount of questions that have piled up in our curious minds. Essentially, git-annex protects your data from accidental modifications and thus keeps it safe. **datalad run** commands mitigate any technical complexity of this completely if `-o/--output` is specified properly, and **datalad unlock** commands can be used to unlock content "by hand" if modifications are performed outside of a **datalad run**.

But there comes the second, tricky part: There are ways to get rid of locking and unlocking within git-annex, using so-called ADJUSTED BRANCHes. This functionality is dependent on the git-annex version one has installed, the git-annex version of the repository, and a use-case dependent comparison of the pros and cons. On Windows systems, this *adjusted mode* is even the *only* mode of operation. In later sections we will see how to use this feature. The next lecture, in any way, will guide us deeper into git-annex, and improve our understanding a slight bit further.

8.2 Data integrity

So far, we mastered quite a number of challenges: Creating and populating a dataset with large and small files, modifying content and saving the changes to history, installing datasets, even as subdatasets within datasets, recording the impact of commands on a dataset with the run and re-run commands, and capturing plenty of PROVENANCE on the way. We further noticed that when we modified content in `notes.txt` or `list_titles.sh`, the modified content was in a *text file*. We learned that this precise type of file, in conjunction with the initial configuration template `text2git` we gave to **datalad create**, is meaningful: As the text file is stored in Git and not git-annex, no content unlocking is necessary. As we saw within the demonstrations of **datalad run**, modifying content of non-text files, such as `.jpgs`, typically requires the additional step of *unlocking* file content, either by hand with the **datalad unlock** command, or within **datalad run** using the `-o/--output` flag.

There is one detail about DataLad datasets that we have not covered yet. Its both a crucial aspect to understanding certain aspects of a dataset, but it is also a potential source of confusion that we want to eradicate.

You might have noticed already that an `ls -l` or `tree` command in your dataset shows small arrows and quite cryptic paths following each non-text file. Maybe your shell also displays these files in a different color than text files when listing them. We'll take a look together, using the books/ directory as an example:

> **W8.1 This will look different to you**
>
> First of all, the `tree` equivalent provided by CONDAs m2-base package doesn't list individual files, only directories. And, secondly, even if you list the individual files (e.g., with `ls -l`), you would not see the SYMLINKs shown below. Due to insufficient support of symlinks on Windows, git-annex does not use them. Please read on for a basic understanding of how git-annex usually works – a Windows Wit at the end of this section will then highlight the difference in functionality on Windows.

```
# in the root of DataLad-101
$ cd books
$ tree
.
├── bash_guide.pdf -> ../.git/annex/objects/WF/Gq/MD5E-s1198170--
→0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
├── byte-of-python.pdf -> ../.git/annex/objects/z1/Q8/MD5E-s4208954--
→ab3a8c2f6b76b18b43c5949e0661e266.pdf/MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
├── progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-s12465653--
→05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.
→pdf
└── TLCL.pdf -> ../.git/annex/objects/jf/3M/MD5E-s2120211--
→06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf

0 directories, 4 files
```

If you do not know what you are looking at, this looks weird, if not worse: intimidating, wrong, or broken. First of all: no, **it is all fine**. But let's start with the basics of what is displayed here to understand it.

The small `->` symbol connecting one path (the book's name) to another path (the weird sequence of characters ending in `.pdf`) is what is called a *symbolic link* (short: SYMLINK) or *softlink*. It is a term for any file that contains a reference to another file or directory as a RELATIVE PATH or ABSOLUTE PATH. If you use Windows, you are familiar with a related, although more basic concept: a shortcut.

This means that the files that are in the locations in which you saved content and are named as you named your files (e.g., `TLCL.pdf`), do *not actually contain your files' content*: they just point to the place where the actual file content resides.

This sounds weird, and like an unnecessary complication of things. But we will get to why this is relevant and useful shortly. First, however, where exactly are the contents of the files you created or saved?

The start of the link path is `../.git`. The section *Create a dataset* (page 34) contained a note that strongly advised that you to not tamper with (or in the worst case, delete) the `.git` repository in the root of any dataset. One reason why you should not do this is because *this* `.git` directory is where all of your file content is actually stored.

But why is that? We have to talk a bit git-annex now in order to understand it[88].

When a file is saved into a dataset to be tracked, by default – that is in a dataset created without any configuration template – DataLad gives this file to git-annex. Exceptions to this behavior can be defined based on

1. file size

2. and/or path/pattern, and thus for example file extensions, or names, or file types (e.g., text files, as with the `text2git` configuration template).

git-annex, in order to version control the data, takes the file content and moves it under `.git/annex/objects` – the so called OBJECT-TREE. It further renames the file into the sequence of characters you can see in the path, and in its place creates a symlink with the original file name,

---

[88] Note, though, that the information below applies to everything that is not an *adjusted branch* in a git-annex *v7 repository* – this information does not make sense yet, but it will be an important reference point later on. Just for the record: Currently, we do not yet have a v7 repository in `DataLad-101`, and the explanation below applies to our current dataset.

　　　　　　　　　　　　　　　　　　　　　　　　Chapter 8. Under the hood: git-annex

pointing to the new location. This process is often referred to as a file being *annexed*, and the object tree is also known as the *annex* of a dataset.

> ### W8.2 What happens on Windows?
>
> Windows has insufficient support for SYMLINKs and revoking write PERMISSIONS on files. Therefore, GIT-ANNEX classifies it as a CRIPPLED FILESYSTEM and has to stray from its default behavior. While git-annex on Unix-based file operating systems stores data in the annex and creates a symlink in the data's original place, on Windows it moves data into the ANNEX and creates a *copy* of the data in its original place.
>
> **Why is that?** Data *needs* to be in the annex for version control and transport logistics – the annex is able to store all previous versions of the data, and manage the transport to other storage locations if you want to publish your dataset. But as the *Findoutmore in this section* (page 89) will show, the ANNEX is a non-human readable tree structure, and data thus also needs to exist in its original location. Thus, it exists in both places: its moved into the annex, and copied back into its original location. Once you edit an annexed file, the most recent version of the file is available in its original location, and past versions are stored and readily available in the annex. If you reset your dataset to a previous state (as is shown in the section *Back and forth in time* (page 245)), the respective version of your data is taken from the annex and copied to replace the newer version, and vice versa.
>
> **But doesn't a copy mean data duplication?** Yes, absolutely! And that is a big downside to DataLad and GIT-ANNEX on Windows. If you have a dataset with annexed file contents (be that a dataset you created and populated yourself, or one that you cloned and got file contents with `datalad get` from), it will take up more space than on a Unix-based system. How much more? Every file that exists in your file hierarchy exists twice. A fresh dataset with one version of each file is thus twice as big as it would be on a Linux computer. Any past version of data does not exist in duplication.
>
> **Step-by-step demonstration**: Let's take a concrete example to explain the last point in more detail. How much space, do you think, is taken up in your dataset by the resized `salt_logo_small.jpg` image? As a reminder: It exists in two versions, a 400 by 400 pixel version (about 250Kb in size), and a 450 by 450 pixel version (about 310Kb in size). The 400 by 400 pixel version is the most recent one. The answer is: about 810Kb (~0.1Mb). The most recent 400x400px version exists twice (in the annex and as a copy), and the 450x450px copy exists once in the annex. If you would reset your dataset to the state when we created the 450x450px version, this file would instead exist twice.
>
> **Can I at least get unused or irrelevant data out of the dataset?** Yes, either with convenience commands (e.g., `git annex unused` followed by `git annex dropunused`), or by explicitly using `drop` on files (or there past versions) that you don't want to keep anymore. Alternatively, you can transfer data you don't need but want to preserve to a different storage location. Later parts of the handbook will demonstrate each of these alternatives.

For a demonstration that this file path is not complete gibberish, take the target path of any of the book's symlinks and open it, for example with `evince <path>`, or any other PDF reader in exchange for `evince`:

```
evince ../.git/annex/objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/
↪MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
```

Even though the path looks cryptic, it works and opens the file. Whenever you use a command

like evince TLCL.pdf, internally, your shell will follow the same cryptic symlink like the one you have just opened.

But *why* does this symlink-ing happen? Up until now, it still seems like a very unnecessary, superfluous thing to do, right?

The resulting symlinks that look like your files but only point to the actual content in .git/ annex/objects are small in size. An ls -lah reveals that all of these symlinks have roughly the same, small size of ~130 Bytes:

```
$ ls -lah
total 24K
drwxr-xr-x 2 adina adina 4.0K Jul 29 16:20 .
drwxr-xr-x 7 adina adina 4.0K Jul 29 16:21 ..
lrwxrwxrwx 1 adina adina  131 Jan 19  2009 bash_guide.pdf -> ../.git/annex/objects/WF/Gq/
↪MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--
↪0ab2c121bcf68d7278af266f6a399c5f.pdf
lrwxrwxrwx 1 adina adina  131 Jun 16  2020 byte-of-python.pdf -> ../.git/annex/objects/z1/
↪Q8/MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf/MD5E-s4208954--
↪ab3a8c2f6b76b18b43c5949e0661e266.pdf
lrwxrwxrwx 1 adina adina  133 Jun 29  2019 progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-
↪s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--
↪05cd7ed561d108c9bcf96022bc78a92c.pdf
lrwxrwxrwx 1 adina adina  131 Jan 28  2019 TLCL.pdf -> ../.git/annex/objects/jf/3M/MD5E-
↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf
```

Here you can see the reason why content is symlinked: Small file size means that *Git can handle those symlinks*! Therefore, instead of large file content, only the symlinks are committed into Git, and the Git repository thus stays lean. Simultaneously, still, all files stored in Git as symlinks can point to arbitrarily large files in the object tree. Within the object tree, git-annex handles file content tracking, and is busy creating and maintaining appropriate symlinks so that your data can be version controlled just as any text file.

This comes with two very important advantages:

One, should you have copies of the same data in different places of your dataset, the symlinks of these files point to the same place (in order to understand why this is the case, you will need to read the hidden section at the end of the page). Therefore, any amount of copies of a piece of data is only one single piece of data in your object tree. This, depending on how much identical file content lies in different parts of your dataset, can save you much disk space and time.

The second advantage is less intuitive but clear for users familiar with Git. Small symlinks can be written very very fast when switching BRANCHes, as opposed to copying and deleting huge data files.

> **G8.1 Speedy branch switches**
>
> Switching branches fast, even when they track vasts amounts of data, lets you work with data with the same routines as in software development.

This leads to a few conclusions:

The first is that you should not be worried to see cryptic looking symlinks in your repository – this is how it should look. You can read the *find-out-more on why these paths look so weird* (page 89) and what all of this has to do with data integrity, if you want to. Its additional information that can help to establish trust in that your data are safely stored and tracked, and

understanding more about the object tree and knowing bits of the git-annex basics can make you more confident in working with your datasets.

The second is that it should now be clear to you why the `.git` directory should not be deleted or in any way modified by hand. This place is where your data are stored, and you can trust git-annex to be better able to work with the paths in the object tree than you or any other human are.

Lastly, understanding that annexed files in your dataset are symlinked will be helpful to understand how common file system operations such as moving, renaming, or copying content translate to dataset modifications in certain situations. Later in this book we will have a section on how to manage the file system in a DataLad dataset (*Miscellaneous file system operations* (page 224)).

> **M8.1 more about paths, checksums, object trees, and data integrity**
>
> So how do these cryptic paths and names in the object tree come into existence? Its not malicious intent that leads to these paths and file names - its checksums.
> When a file is annexed, git-annex generates a *key* (or CHECKSUM) from the **file content**. It uses this key (in part) as a name for the file and as the path in the object tree. Thus, the key is associated with the content of the file (the *value*), and therefore, using this key, file content can be identified – or rather: Based on the keys, it can be identified whether file content changed, and whether two files have identical contents.
> The key is generated using *hashes*. A hash is a function that turns an input (e.g., a PDF file) into a string of characters with a fixed length based on its contents.
> Importantly, a hash function will generate the same character sequence for the same file content, and once file content changes, the generated hash changes, too. Basing the file name on its contents thus becomes a way of ensuring data integrity: File content can not be changed without git-annex noticing, because file's hash, and thus its key in its symlink, will change. Furthermore, if two files have identical hashes, the content in these files is identical. Consequently, if two files have the same symlink, and thus link the same file in the object-tree, they are identical in content. This can save disk space if a dataset contains many identical files: Copies of the same data only need one instance of that content in the object tree, and all copies will symlink to it. If you want to read more about the computer science basics about hashes check out the Wikipedia page here[83].
>
> ```
> # take a look at the last part of the target path:
> $ ls -lah TLCL.pdf
> lrwxrwxrwx 1 adina adina 131 Jan 28  2019 TLCL.pdf -> ../.git/annex/objects/jf/3M/
> ↪MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
> ↪06d1efcb05bb2c55cd039dab3fb28455.pdf
> ```
>
> Let's take a closer look at the structure of the symlink. The key from the hash function is the last part of the name of the file the symlink links to (in which the actual data content is stored).
>
> ```
> # compare it to the checksum (here of type md5sum) of the PDF file and the␣
> ↪subdirectory name
> $ md5sum TLCL.pdf
> 06d1efcb05bb2c55cd039dab3fb28455  TLCL.pdf
> ```
>
> The extension (e.g., .pdf) is appended because some operating systems (*ehem*, Windows) need this information in order to select the right software to open a file. Right at the beginning, the symlink starts with two directories just after `.git/annex/objects/`, consisting of two letters each. These two letters are derived from the md5sum of the

key, and their sole purpose to exist is to avoid issues with too many files in one directory (which is a situation that certain file systems have problems with). The next subdirectory in the symlink helps to prevent accidental deletions and changes, as it does not have write PERMISSIONS, so that users cannot modify any of its underlying contents. This is the reason that annexed files need to be unlocked prior to modifications, and this information will be helpful to understand some file system management operations such as removing files or datasets (see section *Miscellaneous file system operations* (page 224)).

The next part of the symlink contains the actual hash. There are different hash functions available. Depending on which is used, the resulting CHECKSUM has a certain length and structure, and the first part of the symlink actually states which hash function is used. By default, DataLad uses MD5E checksums (relatively short and with a file extension), but should you want to, you can change this default to one of many other types[84]. The reason why MD5E is used is because of its short length – thus it is possible to ensure cross-platform compatibility and share datasets also with users on operating systems that have restrictions on total path lengths, such as Windows.

The one remaining unidentified bit in the file name is the one after the checksum identifier. This part is the size of the content in bytes. An annexed file in the object tree thus has a file name following this structure:

checksum-identifier - size -- checksum . extension

You now know a great deal more about git-annex and the object tree. Maybe you are as amazed as we are about some of the ingenuity used behind the scenes. Even more mesmerizing things about git-annex can be found in its documentation[85].

---

[83] https://en.wikipedia.org/wiki/Hash_function
[84] https://git-annex.branchable.com/backends/
[85] https://git-annex.branchable.com/git-annex/

Broken symlinks

Whenever a symlink points to a non-existent target, this symlink is called *broken*, and opening the symlink would not work as it does not resolve. The section *Miscellaneous file system operations* (page 224) will give a thorough demonstration of how symlinks can break, and how one can fix them again. Even though *broken* sounds troublesome, most types of broken symlinks you will encounter can be fixed, or are not problematic. At this point, you actually have already seen broken symlinks: Back in section *Install datasets* (page 46) we explored the file hierarchy in an installed subdataset that contained many annexed mp3 files. Upon the initial **datalad clone**, the annexed files were not present locally. Instead, their symlinks (stored in Git) existed and allowed to explore which file's contents could be retrieved. These symlinks point to nothing, though, as the content isn't yet present locally, and are thus *broken*. This state, however, is not problematic at all. Once the content is retrieved via **datalad get**, the symlink is functional again.

Nevertheless, it may be important to know that some tools that you would expect to work in a dataset with not yet retrieved file contents can encounter unintuitive problems. Some **file managers** (e.g., OSX's Finder) may not display broken symlinks. In these cases, it will be impossible to browse and explore the file hierarchy of not-yet-retrieved files with the file manager. You can make sure to always be able to see the file hierarchy in two separate ways: Upgrade your file manager to display file types in a DataLad datasets (e.g., the git-annex-turtle extension[86] for Finder). Alternatively, use the **ls** command in a terminal instead of a file manager

---

[86] https://github.com/andrewringler/git-annex-turtle

GUI. Other tools may be more more specialized, smaller, or domain-specific, and may fail to correctly work with broken symlinks, or display unhelpful error messages when handling them, or require additional flags to modify their behavior (such as the *BIDS Validator* (page 487), used in the neuroimaging community). When encountering unexpected behavior or failures, try to keep in mind that a dataset without retrieved content appears to be a pile of broken symlinks to a range of tools, consult a tools documentation with regard to symlinks, and check whether data retrieval fixes persisting problems.

Finally, if you are still in the books/ directory, go back into the root of the superdataset.

```
$ cd ../
```

Cross-OS filesharing with symlinks (WSL2 only)

Are you using DataLad on the Windows Subsystem for Linux? If so, please take a look into the Windows Wit below.

> **W8.3 Accessing symlinked files from your Windows system**
>
> If you are using WSL2 you have access to a Linux kernel and POSIX filesystem, including symlink support. Your DataLad experience has therefore been exactly as it has been for macOS or Linux users. But one thing that bears the need for additional information is sharing files in dataset between your Linux and Windows system.
>
> Its fantastic that files created under Linux can be shared to Windows and used by Windows tools. Usually, you should be able to open an explorer and type `\\wsl$\<distro>\<path>` in the address bar to navigate to files under Linux, or type `explorer.exe` into the WSL2 terminal. Some core limitations of Windows can't be overcome, though: Windows usually isn't capable of handling symlinks. So while WSL2 can expose your dataset filled with symlinked files to Windows, your Windows tools can fail to open them. How can this be fixed?
>
> Whenever you need to work with files from your datasets under Windows, you should *unlock* with `datalad unlock`. This operation copies the file from the annex back to its original location, and thus removes the symlink (and also returns write PERMISSIONS to the file). Alternatively, use git-annex adjust –unlock[87] to switch to a new dataset BRANCH in which all files are unlocked. The branch is called `adjusted/<branchname>(unlocked)` (e.g., if the original branch name was `main`, the new, adjusted branch will be called `adjusted/main(unlocked)`). You can switch back to your original branch using `git checkout <branchname>`.
>
> ---
> [87] https://git-annex.branchable.com/git-annex-adjust/

COLLABORATION
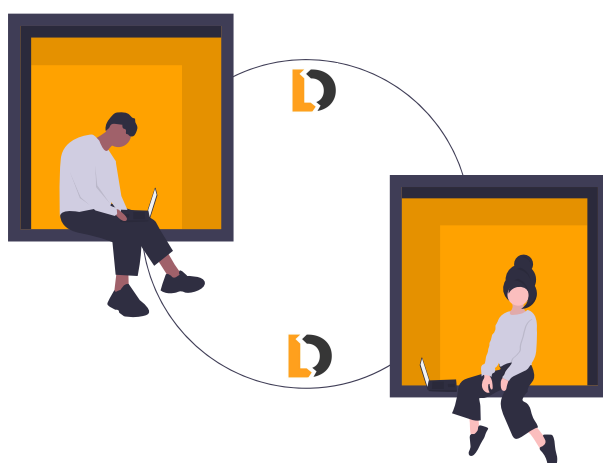
9.1 Looking without touching

Only now, several weeks into the DataLad-101 course does your room mate realize that he has enrolled in the course as well, but has not yet attended at all. "Oh man, can you help me catch up?" he asks you one day. "Sharing just your notes would be really cool for a start already!"

"Sure thing", you say, and decide that it's probably best if he gets all of the `DataLad-101` course dataset. Sharing datasets was something you wanted to look into soon, anyway.

This is one exciting aspect of DataLad datasets that has yet been missing from this course: How does one share a dataset? In this section, we will cover the simplest way of sharing a dataset: on a local or shared file system, via an *installation* with a path as a source.

⚠ **More on public data sharing**

Interested in sharing datasets *publicly*? Read this chapter to get a feel for all relevant basic concepts of sharing datasets. Afterwards, head over to chapter *Third party infrastructure* (page 177) to find out how to share a dataset on third-party infrastructure.

In this scenario multiple people can access the very same files at the same time, often on the same machine (e.g., a shared workstation, or a server that people can "SSH" into). You might think: "What do I need DataLad for, if everyone can already access everything?" However, universal, unrestricted access can easily lead to chaos. DataLad can help facilitate collaboration without requiring ultimate trust and reliability of all participants. Essentially, with a shared dataset, collaborators can look and use your dataset without ever touching it.

To demonstrate how to share a DataLad dataset on a common file system, we will pretend that your personal computer can be accessed by other users. Let's say that your room mate has access, and you're making sure that there is a DataLad-101 dataset in a different place on the file system for him to access and work with.

This is indeed a common real-world use case: Two users on a shared file system sharing a dataset with each other. But as we can not easily simulate a second user in this handbook, for now, you will have to share your dataset with yourself. This endeavor serves several purposes: For one, you will experience a very easy way of sharing a dataset. Secondly, it will show you how a dataset can be obtained from a path (instead of a URL as shown in the section *Install datasets* (page 46)). Thirdly, DataLad-101 is a dataset that can showcase many different properties of a dataset already, but it will be an additional learning experience to see how the different parts of the dataset – text files, larger files, datalad subdataset, **datalad run** commands – will appear upon installation when shared. And lastly, you will likely "share a dataset with yourself" whenever you will be using a particular dataset of your own creation as input for one or more projects.

"Awesome!" exclaims your room mate as you take out your Laptop to share the dataset. "You're really saving my ass here. I'll make up for it when we prepare for the final", he promises.

To install DataLad-101 into a different part of your file system, navigate out of DataLad-101, and – for simplicity – create a new directory, mock_user, right next to it:

```
$ cd ../
$ mkdir mock_user
```

For simplicity, pretend that this is a second user's – your room mate's – home directory. Furthermore, let's for now disregard anything about PERMISSIONS. In a real-world example you likely would not be able to read and write to a different user's directories, but we will talk about permissions later.

After creation, navigate into mock_user and install the dataset DataLad-101. To do this, use **datalad clone**, and provide a path to your original dataset. Here is how it looks like:

```
$ cd mock_user
$ datalad clone ../DataLad-101 --description "DataLad-101 in mock_user"
[INFO] Cloning dataset to Dataset(/home/me/dl-101/mock_user/DataLad-101)
[INFO] Attempting to clone from ../DataLad-101 to /home/me/dl-101/mock_user/DataLad-101
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/mock_user/DataLad-101)
[INFO] scanning for unlocked files (this may take some time)
install(ok): /home/me/dl-101/mock_user/DataLad-101 (dataset)
```

This will install your dataset DataLad-101 into your room mate's home directory. Note that we have given this new dataset a description about its location as well. Note further that we have not provided the optional destination path to **datalad clone**, and hence it installed the dataset under its original name in the current directory.

Together with your room mate, you go ahead and see what this dataset looks like. Before running the command, try to predict what you will see.

```
$ cd DataLad-101
$ tree
.
├── books
│   ├── bash_guide.pdf -> ../.git/annex/objects/WF/Gq/MD5E-s1198170--
→0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
```

```
|   ├── byte-of-python.pdf -> ../.git/annex/objects/z1/Q8/MD5E-s4208954--
↪ab3a8c2f6b76b18b43c5949e0661e266.pdf/MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
|   ├── progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-s12465653--
↪05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.
↪pdf
|   └── TLCL.pdf -> ../.git/annex/objects/jf/3M/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
├── code
|   └── list_titles.sh
├── notes.txt
└── recordings
    ├── interval_logo_small.jpg -> ../.git/annex/objects/jx/KK/MD5E-s100593--
↪c4b4290cb5d616154e80cddee76cb756.jpg/MD5E-s100593--c4b4290cb5d616154e80cddee76cb756.jpg
    ├── longnow
    ├── podcasts.tsv
    └── salt_logo_small.jpg -> ../.git/annex/objects/xJ/4G/MD5E-s260607--
↪4e695af0f3e8e836fcfc55f815940059.jpg/MD5E-s260607--4e695af0f3e8e836fcfc55f815940059.jpg

4 directories, 9 files
```

There are a number of interesting things, and your room mate is the first to notice them:

"Hey, can you explain some things to me?", he asks. "This directory here, "longnow", why is it empty?" True, the subdataset has a directory name but apart from this, the longnow directory appears empty.

"Also, why do the PDFs in books/ and the .jpg files appear so weird? They have this cryptic path right next to them, and look, if I try to open one of them, it fails! Did something go wrong when we installed the dataset?" he worries. Indeed, the PDFs and pictures appear just as they did in the original dataset on first sight: They are symlinks pointing to some location in the object tree. To reassure your room mate that everything is fine you quickly explain to him the concept of a symlink and the OBJECT-TREE of GIT-ANNEX.

"But why does the PDF not open when I try to open it?" he repeats. True, these files cannot be opened. This mimics our experience when installing the longnow subdataset: Right after installation, the .mp3 files also could not be opened, because their file content was not yet retrieved. You begin to explain to your room mate how DataLad retrieves only minimal metadata about which files actually exist in a dataset upon a **datalad clone**. "It's really handy", you tell him. "This way you can decide which book you want to read, and then retrieve what you need. Everything that is *annexed* is retrieved on demand. Note though that the text files contents are present, and the files can be opened – this is because these files are stored in GIT. So you already have my notes, and you can decide for yourself whether you want to get the books."

To demonstrate this, you decide to examine the PDFs further. "Try to get one of the books", you instruct your room mate:

```
$ datalad get books/progit.pdf
get(ok): books/progit.pdf (file) [from origin...]
```

"Opening this file will work, because the content was retrieved from the original dataset.", you explain, proud that this worked just as you thought it would. Your room mate is excited by this magical command. You however begin to wonder: how does DataLad know where to look for that original content?

This information comes from git-annex. Before getting the next PDF, let's query git-annex where

its content is stored:

```
$ git annex whereis books/TLCL.pdf
whereis books/TLCL.pdf (1 copy)
        142a8659-5566-466d-af37-5958d071a995 -- me@muninn:~/dl-101/DataLad-101 [origin]
ok
```

Oh, another SHASUM! This time however not in a symlink... "That's hard to read – what is it?" your room mate asks. You can recognize a path to the dataset on your computer, prefixed with the user and hostname of your computer. "This", you exclaim, excited about your own realization, "is my dataset's location I'm sharing it from!"

> **M9.1 What is this location, and what if I provided a description?**
>
> Back in the very first section of the Basics, *Create a dataset* (page 34), a hidden section mentioned the --description option of **datalad create**. With this option, you can provide a description about the *location* of your dataset.
> The **git annex whereis** command, finally, is where such a description can become handy: If you had created the dataset with
>
> ```
> $ datalad create --description "course on DataLad-101 on my private Laptop" -c␣
> ↪text2git DataLad-101
> ```
>
> the command would show course on DataLad-101 on my private Laptop after the SHASUM – and thus a more human-readable description of *where* file content is stored. This becomes especially useful when the number of repository copies increases. If you have only one other dataset it may be easy to remember what and where it is. But once you have one back-up of your dataset on a USB-Stick, one dataset shared with Dropbox, and a third one on your institutions GITLAB instance you will be grateful for the descriptions you provided these locations with.
> The current report of the location of the dataset is in the format user@host:path. As one computer this book is being build on is called "muninn" and its user "me", it could look like this: me@muninn:~/dl-101/DataLad-101.
> If the physical location of a dataset is not relevant, ambiguous, or volatile, or if it has an ANNEX that could move within the foreseeable lifetime of a dataset, a custom description with the relevant information on the dataset is superior. If this is not the case, decide for yourself whether you want to use the --description option for future datasets or not depending on what you find more readable – a self-made location description, or an automatic user@host:path information.

The message further informs you that there is only "(1 copy)" of this file content. This makes sense: There is only your own, original DataLad-101 dataset in which this book is saved.

To retrieve file content of an annexed file such as one of these PDFs, git-annex will try to obtain it from the locations it knows to contain this content. It uses the checksums to identify these locations. Every copy of a dataset will get a unique ID with such a checksum. Note however that just because git-annex knows a certain location where content was once it does not guarantee that retrieval will work. If one location is a USB-Stick that is in your bag pack instead of your USB port, a second location is a hard drive that you deleted all of its previous contents (including dataset content) from, and another location is a web server, but you are not connected to the internet, git-annex will not succeed in retrieving contents from these locations. As long as there is at least one location that contains the file and is accessible, though, git-annex will get the content. Therefore, for the books in your dataset, retrieving contents works because you and your room mate share the same file system. If you'd share the dataset with anyone without

access to your file system, `datalad get` would not work, because it can not access your files.

But there is one book that does not suffer from this restriction: The `bash_guide.pdf`. This book was not manually downloaded and saved to the dataset with `wget` (thus keeping DataLad in the dark about where it came from), but it was obtained with the **`datalad download-url`** command. This registered the books original source in the dataset, and here is why that is useful:

```
$ git annex whereis books/bash_guide.pdf
whereis books/bash_guide.pdf (2 copies)
        00000000-0000-0000-0000-000000000001 -- web
        142a8659-5566-466d-af37-5958d071a995 -- me@muninn:~/dl-101/DataLad-101 [origin]

  web: http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf
ok
```

Unlike the `TLCL.pdf` book, this book has two sources, and one of them is web. The second to last line specifies the precise URL you downloaded the file from. Thus, for this book, your room mate is always able to obtain it (as long as the URL remains valid), even if you would delete your `DataLad-101` dataset. Quite useful, this provenance, right?

Let's now turn to the fact that the subdataset `longnow` contains neither file content nor file metadata information to explore the contents of the dataset: there are no subdirectories or any files under `recordings/longnow/`. This is behavior that you have not observed until now.

To fix this and obtain file availability metadata, you have to run a somewhat unexpected command:

```
$ datalad get -n recordings/longnow
[INFO] Cloning dataset to Dataset(/home/me/dl-101/mock_user/DataLad-101/recordings/
↪longnow)
[INFO] Attempting to clone from /home/me/dl-101/DataLad-101/recordings/longnow to /home/
↪me/dl-101/mock_user/DataLad-101/recordings/longnow
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/mock_user/DataLad-101/
↪recordings/longnow)
[INFO] scanning for unlocked files (this may take some time)
install(ok): /home/me/dl-101/mock_user/DataLad-101/recordings/longnow (dataset)␣
↪[Installed subdataset in order to get /home/me/dl-101/mock_user/DataLad-101/recordings/
↪longnow]
```

The section below will elaborate on **`datalad get`** and the `-n`/`--no-data` option, but for now, let's first see what has changed after running the above command (excerpt):

```
$ tree
.
├── books
│   ├── bash_guide.pdf -> ../.git/annex/objects/WF/Gq/MD5E-s1198170--
↪0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
│   ├── byte-of-python.pdf -> ../.git/annex/objects/z1/Q8/MD5E-s4208954--
↪ab3a8c2f6b76b18b43c5949e0661e266.pdf/MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
│   ├── progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-s12465653--
↪05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.
↪pdf
│   └── TLCL.pdf -> ../.git/annex/objects/jf/3M/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
├── code
│   └── list_titles.sh
├── notes.txt
```

(continues on next page)

```
└── recordings
    ├── interval_logo_small.jpg -> ../.git/annex/objects/jx/KK/MD5E-s100593--
↪c4b4290cb5d616154e80cddee76cb756.jpg/MD5E-s100593--c4b4290cb5d616154e80cddee76cb756.jpg
    ├── longnow
        ├── Long_Now__Conversations_at_The_Interval
            ├── 2017_06_09__How_Digital_Memory_Is_Shaping_Our_Future__Abby_Smith_Rumsey.
↪mp3 -> ../.git/annex/objects/8j/kQ/MD5E-s66305442--c723d53d207e6d82dd64c3909a6a93b0.mp3/
↪MD5E-s66305442--c723d53d207e6d82dd64c3909a6a93b0.mp3
        │   ├── 2017_06_09__Pace_Layers_Thinking__Stewart_Brand__Paul_Saffo.mp3 -> ../.
↪git/annex/objects/Qk/9M/MD5E-s112801659--00a42a1a617485fb2c03cbf8482c905c.mp3/MD5E-
↪s112801659--00a42a1a617485fb2c03cbf8482c905c.mp3
        │   ├── 2017_06_09__Proof__The_Science_of_Booze__Adam_Rogers.mp3 -> ../.git/annex/
↪objects/FP/96/MD5E-s60091960--6e48eceb5c54d458164c2d0f47b540bc.mp3/MD5E-s60091960--
↪6e48eceb5c54d458164c2d0f47b540bc.mp3
        │   ├── 2017_06_09__Seveneves_at_The_Interval__Neal_Stephenson.mp3 -> ../.git/
↪annex/objects/Wf/5Q/MD5E-s66431897--aff90c838a1c4a363bb9d83a46fa989b.mp3/MD5E-s66431897-
↪-aff90c838a1c4a363bb9d83a46fa989b.mp3
        │   ├── 2017_06_09__Talking_with_Robots_about_Architecture__Jeffrey_McGrew.mp3 ->␣
↪../.git/annex/objects/Fj/9V/MD5E-s61491081--c4e88ea062c0afdbea73d295922c5759.mp3/MD5E-
↪s61491081--c4e88ea062c0afdbea73d295922c5759.mp3
        │   ├── 2017_06_09__The_Red_Planet_for_Real__Andy_Weir.mp3 -> ../.git/annex/
↪objects/xq/Q3/MD5E-s136924472--0d1072105caa56475df9037670d35a06.mp3/MD5E-s136924472--
↪0d1072105caa56475df9037670d35a06.mp3
```

Interesting! The file metadata information is now present, and we can explore the file hierarchy. The file content, however, is not present yet.

What has happened here?

When DataLad installs a dataset, it will by default only obtain the superdataset, and not any subdatasets. The superdataset contains the information that a subdataset exists though – the subdataset is *registered* in the superdataset. This is why the subdataset name exists as a directory. A subsequent `datalad get -n path/to/longnow` will install the registered subdataset again, just as we did in the example above.

But what about the `-n` option for `datalad get`? Previously, we used `datalad get` to get file content. However, `get` can operate on more than just the level of *files* or *directories*. Instead, it can also operate on the level of *datasets*. Regardless of whether it is a single file (such as books/ `TLCL.pdf`) or a registered subdataset (such as recordings/longnow), `get` will operate on it to 1) install it – if it is a not yet installed subdataset – and 2) retrieve the contents of any files. That makes it very easy to get your file content, regardless of how your dataset may be structured – it is always the same command, and DataLad blurs the boundaries between superdatasets and subdatasets.

In the above example, we called `datalad get` with the option `-n/--no-data`. This option prevents that `get` obtains the data of individual files or directories, thus limiting its scope to the level of datasets as only a `datalad clone` is performed. Without this option, the command would have retrieved all of the subdatasets contents right away. But with `-n/--no-data`, it only installed the subdataset to retrieve the meta data about file availability.

To explicitly install all potential subdatasets *recursively*, that is, all of the subdatasets inside it as well, one can give the `-r/--recursive` option to `get`:

```
datalad get -n -r <subds>
```

This would install the subds subdataset and all potential further subdatasets inside of it, and

---

the meta data about file hierarchies would have been available right away for every subdataset inside of subds. If you had several subdatasets and would not provide a path to a single dataset, but, say, the current directory (. as in **datalad get -n -r .**), it would clone all registered subdatasets recursively.

So why is a recursive get not the default behavior? In *Dataset nesting* (page 53) we learned that datasets can be nested *arbitrarily* deep. Upon getting the meta data of one dataset you might not want to also install a few dozen levels of nested subdatasets right away.

However, there is a middle way[89]: The `--recursion-limit` option let's you specify how many levels of subdatasets should be installed together with the first subdataset:

```
datalad get -n -r --recursion-limit 1 <subds>
```

> **M9.2 datalad clone versus datalad install**
>
> You may remember from section *Install datasets* (page 46) that DataLad has two commands to obtain datasets, **datalad clone** and **datalad install**. The command structure of **install** and **datalad clone** are almost identical:
>
> ```
> $ datalad install [-d/--dataset PATH] [-D/--description] --source PATH/URL [DEST-
> →PATH ...]
> $ datalad clone [-d/--dataset PATH] [-D/--description] SOURCE-PATH/URL [DEST-PATH]
> ```
>
> Both commands are also often interchangeable: To create a copy of your DataLad-101 dataset for your roommate, or to obtain the longnow subdataset in section *Install datasets* (page 46) you could have used **datalad install** as well. From a user's perspective, the only difference is whether you'd need -s/--source in the command call:
>
> ```
> $ datalad install --source ../DataLad-101
> # versus
> $ datalad clone ../DataLad-101
> ```
>
> On a technical layer, **datalad clone** is a subset (or rather: the underlying function) of the **datalad install** command. Whenever you use **datalad install**, it will call **datalad clone** underneath the hood. **datalad install**, however, adds to **datalad clone** in that it has slightly more complex functionality. Thus, while command structure is more intuitive, the capacities of **clone** are also slightly more limited than those of **install** in comparison. Unlike **datalad clone**, **datalad install** provides a -r/--recursive operation, i.e., it can obtain (clone) a dataset and potential subdatasets right at the time of superdataset installation. You can pick for yourself which command you are more comfortable with. In the handbook, we use **clone** for its more intuitive behavior, but you will often note that we use the terms "installed dataset" and "cloned dataset" interchangeably.

To summarize what you learned in this section, write a note on how to install a dataset using a path as a source on a common file system.

Write this note in "your own" (the original) DataLad-101 dataset, though!

---

[89] Another alternative to a recursion limit to **datalad get -n -r** is a dataset configuration that specifies subdatasets that should *not* be cloned recursively, unless explicitly given to the command with a path. With this configuration, a superdataset's maintainer can safeguard users and prevent potentially large amounts of subdatasets to be cloned. You can learn more about this configuration in the section *More on DIY configurations* (page 117).

```
# navigate back into the original dataset
$ cd ../../DataLad-101
# write the note
$ cat << EOT >> notes.txt
A source to install a dataset from can also be a path, for example as
in "datalad clone ../DataLad-101".

Just as in creating datasets, you can add a description on the
location of the new dataset clone with the -D/--description option.

Note that subdatasets will not be installed by default, but are only
registered in the superdataset -- you will have to do a
"datalad get -n PATH/TO/SUBDATASET" to install the subdataset for file
availability meta data. The -n/--no-data options prevents that file
contents are also downloaded.

Note that a recursive "datalad get" would install all further
registered subdatasets underneath a subdataset, so a safer way to
proceed is to set a decent --recursion-limit:
"datalad get -n -r --recursion-limit 2 <subds>"

EOT
```

Save this note.

```
$ datalad save -m "add note about cloning from paths and recursive datalad get"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

> **G9.1 Get a clone**
>
> A dataset that is installed from an existing source, e.g., a path or URL, is the DataLad
> equivalent of a *clone* in Git.

9.2 Where's Waldo?

So far, you and your room mate have created a copy of the DataLad-101 dataset on the same
file system but a different place by installing it from a path.

You have observed that the -r/--recursive option needs to be given to **datalad get [-n/
--no-data]** in order to install further potential subdatasets in one go. Only then is the sub-
datasets file content availability metadata present to explore the file hierarchy available within
the subdataset. Alternatively, a **datalad get -n <subds>** takes care of installing exactly the
specified registered subdataset.

And you have mesmerized your room mate by showing him how GIT-ANNEX retrieved large file
contents from the original dataset.

Let's now see the **git annex whereis** command in more detail, and find out how git-annex
knows *where* file content can be obtained from. Within the original DataLad-101 dataset, you

retrieved some of the .mp3 files via **datalad get**, but not others. How will this influence the output of **git annex whereis**, you wonder?

Together with your room mate, you decide to find out. You navigate back into the installed dataset, and run **git annex whereis** on a file that you once retrieved file content for, and on a file that you did not yet retrieve file content for. Here is the output for the retrieved file:

```
# navigate back into the clone of DataLad-101
$ cd ../mock_user/DataLad-101
# navigate into the subdirectory
$ cd recordings/longnow
# file content exists in original DataLad-101 for this file
$ git annex whereis Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__
↪The_Long_Now.mp3
whereis Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.
↪mp3 (3 copies)
        00000000-0000-0000-0000-000000000001 -- web
         da3bf937-5bd2-43ea-a07b-bcbe71f3b875 -- mih@medusa:/tmp/seminars-on-longterm-
↪thinking
        eb47bf12-2366-495d-80a6-2861eb665f06 -- me@muninn:~/dl-101/DataLad-101/recordings/
↪longnow [origin]

  web: http://podcast.longnow.org/salt/redirect/salt-020031114-eno-podcast.mp3
ok
```

And here is the output for a file that you did not yet retrieve content for in your original DataLad-101 dataset.

```
# but not for this:
$ git annex whereis Long_Now__Seminars_About_Long_term_Thinking/2005_01_15__James_Carse__
↪Religious_War_In_Light_of_the_Infinite_Game.mp3
whereis Long_Now__Seminars_About_Long_term_Thinking/2005_01_15__James_Carse__Religious_
↪War_In_Light_of_the_Infinite_Game.mp3 (2 copies)
        00000000-0000-0000-0000-000000000001 -- web
         da3bf937-5bd2-43ea-a07b-bcbe71f3b875 -- mih@medusa:/tmp/seminars-on-longterm-
↪thinking

  web: http://podcast.longnow.org/salt/redirect/salt-020050114-carse-podcast.mp3
ok
```

As you can see, the file content previously downloaded with a **datalad get** has a third source, your original dataset on your computer. The file we did not yet retrieve in the original dataset only has only two sources.

Let's see how this affects a **datalad get**:

```
# get the first file
$ datalad get Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_
↪Now.mp3
get(ok): Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.
↪mp3 (file) [from origin...]
```

```
# get the second file
$ datalad get Long_Now__Seminars_About_Long_term_Thinking/2005_01_15__James_Carse__
↪Religious_War_In_Light_of_the_Infinite_Game.mp3
get(ok): Long_Now__Seminars_About_Long_term_Thinking/2005_01_15__James_Carse__Religious_
↪War_In_Light_of_the_Infinite_Game.mp3 (file) [from web...]
```

The most important thing to note is: It worked in both cases, regardless of whether the original `DataLad-101` dataset contained the file content or not.

We can see that git-annex used two different sources to retrieve the content from, though, if we look at the very end of the `get` summary. The first file was retrieved "`from origin...`". `Origin` is Git terminology for "from where the dataset was copied from" – `origin` therefore is the original `DataLad-101` dataset.

The second file was retrieved "`from web...`", and thus from a different source. This source is called web because it actually is a URL through which this particular podcast-episode is made available in the first place. You might also have noticed that the download from web took longer than the retrieval from the directory on the same file system. But we will get into the details of this type of content source once we cover the `importfeed` and `add-url` functions[90].

Let's for now add a note on the **git annex whereis** command. Again, do this in the original `DataLad-101` directory, and do not forget to save it.

```
# navigate back:
$ cd ../../../../DataLad-101

# write the note
$ cat << EOT >> notes.txt
The command "git annex whereis PATH" lists the repositories that have
the file content of an annexed file. When using "datalad get" to
retrieve file content, those repositories will be queried.

EOT

$ datalad status
 modified: notes.txt (file)

$ datalad save -m "add note on git annex whereis"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

9.3 Retrace and reenact

"Thanks a lot for sharing your dataset with me! This is super helpful. I'm sure I'll catch up in no time!", your room mate says confidently. "How far did you get with the DataLad commands yet?" he asks at last.

"Mhh, I think the last big one was **datalad run**. Actually, let me quickly show you what this command does. There is something that I've been wanting to try anyway." you say.

The dataset you shared contained a number of **datalad run** commands. For example, you created the simple `podcasts.tsv` file that listed all titles and speaker names of the longnow podcasts.

---

[90] Maybe you wonder what the location `mih@medusa` is. It is a copy of the data on an account belonging to user `mih` on the host name `medusa`. Because we do not have the host names' address, nor log-in credentials for this user, we can not retrieve content from this location. However, somebody else (for example the user `mih`) could.

Given that you learned to create "proper" `datalad run` commands, complete with `--input` and `--output` specification, anyone should be able to `datalad rerun` these commits easily. This is what you want to try now.

You begin to think about which `datalad run` commit would be the most useful one to take a look at. The creation of `podcasts.tsv` was a bit dull – at this point in time, you didn't yet know about `--input` and `--output` arguments, and the resulting output is present anyway because text files like this `.tsv` file are stored in Git. However, one of the attempts to resize a picture could be useful. The input, the podcast logos, is not yet retrieved, nor is the resulting, resized image. "Let's go for this!", you say, and drag your confused room mate to the computer screen.

First of all, find the commit shasum of the command you want to run by taking a look into the history of the dataset (in the shared dataset):

```
# navigate into the shared copy
$ cd ../mock_user/DataLad-101


# lets view the history
$ git log --oneline -n 10
41ea4e6 add note on clean datasets
4753f3f [DATALAD RUNCMD] Resize logo for slides
4951771 [DATALAD RUNCMD] Resize logo for slides
926a797 add additional notes on run options
4cf3515 [DATALAD RUNCMD] convert -resize 450x450 recordings/longn...
19a9a7d resized picture by hand
91aeee9 [DATALAD RUNCMD] convert -resize 400x400 recordings/longn...
da12c3f add note on basic datalad run and datalad rerun
355446e add note datalad and git diff
d5edb18 [DATALAD RUNCMD] create a list of podcast titles
```

Ah, there it is, the second most recent commit. Just as already done in section *DataLad, Re-Run!* (page 64), take this shasum and plug it into a `datalad rerun` command:

```
$ datalad rerun 4753f3f58f6cb9b2a70f71ad56fd0173c453853f
[INFO] run commit 4753f3f; (Resize logo for s...)
[INFO] Making sure inputs are available (this may take some time)
get(ok): recordings/longnow/.datalad/feed_metadata/logo_salt.jpg (file) [from origin...]
[WARNING] no content present; cannot unlock [unlock(/home/me/dl-101/mock_user/DataLad-101/
↪recordings/salt_logo_small.jpg)]
remove(ok): recordings/salt_logo_small.jpg
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): recordings/salt_logo_small.jpg (file)
action summary:
  add (ok: 1)
  get (notneeded: 1, ok: 1)
  remove (ok: 1)
  save (notneeded: 2)
```

"This was so easy!" you exclaim. DataLad retrieved the missing file content from the subdataset and it tried to unlock the output prior to the command execution. Note that because you did not retrieve the output, `recordings/salt_logo_small.jpg`, yet, the missing content could not be unlocked. DataLad warns you about this, but proceeds successfully.

Your room mate now not only knows how exactly the resized file came into existence, but he can also reproduce your exact steps to create it. "This is as reproducible as it can be!" you think in awe.

9.4 Stay up to date

All of what you have seen about sharing dataset was really cool, and for the most part also surprisingly intuitive. **datalad run** commands or file retrieval worked exactly as you imagined it to work, and you begin to think that slowly but steadily you're getting a feel about how DataLad really works.

But to be honest, so far, sharing the dataset with DataLad was also remarkably unexciting given that you already knew most of the dataset magic that your room mate currently is still mesmerized about. To be honest, you're not yet certain whether sharing data with DataLad really improves your life up until this point. After all, you could have just copied your directory into your mock_user directory and this would have resulted in about the same output, right?

What we will be looking into now is how shared DataLad datasets can be updated.

Remember that you added some notes on **datalad clone**, **datalad get**, and **git annex whereis** into the original DataLad-101?

This is a change that is not reflected in your "shared" installation in ../mock_user/DataLad-101:

```
# Inside the installed copy, view the last 15 lines of notes.txt
$ tail notes.txt
should be specified with an -o/--output flag. Upon a run or rerun of
the command, the contents of these files will get unlocked so that
they can be modified.

Important! If the dataset is not "clean" (a datalad status output is
empty), datalad run will not work - you will have to save
modifications present in your dataset.
A suboptimal alternative is the --explicit flag, used to record only
those changes done to the files listed with --output flags.
```

But the original intention of sharing the dataset with your room mate was to give him access to your notes. How does he get the notes that you have added in the last two sections, for example?

This installed copy of DataLad-101 knows its origin, i.e., the place it was installed from. Using this information, it can query the original dataset whether any changes happened since the last time it checked, and if so, retrieve and integrate them.

This is done with the **datalad update --merge** command (datalad-update manual).

```
$ datalad update --merge
[INFO] Fetching updates for Dataset(/home/me/dl-101/mock_user/DataLad-101)
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
merge(ok): . (dataset) [Merged origin/master]
update(ok): . (dataset)
action summary:
  merge (ok: 1)
  update (ok: 1)
```

Importantly, run this command either within the specific (sub)dataset you are interested in, or provide a path to the root of the dataset you are interested in with the -d/--dataset flag. If

you would run the command within the `longnow` subdataset, you would query this subdatasets' `origin` for updates, not the original `DataLad-101` dataset.

Let's check the contents in `notes.txt` to see whether the previously missing changes are now present:

```
# view the last 15 lines of notes.txt
$ tail notes.txt

Note that a recursive "datalad get" would install all further
registered subdatasets underneath a subdataset, so a safer way to
proceed is to set a decent --recursion-limit:
"datalad get -n -r --recursion-limit 2 <subds>"

The command "git annex whereis PATH" lists the repositories that have
the file content of an annexed file. When using "datalad get" to
retrieve file content, those repositories will be queried.
```

Wohoo, the contents are here!

Therefore, sharing DataLad datasets by installing them enables you to update the datasets content should the original datasets' content change – in only a single command. How cool is that?!

Conclude this section by adding a note about updating a dataset to your own `DataLad-101` dataset:

```
# navigate back:
$ cd ../../DataLad-101

# write the note
$ cat << EOT >> notes.txt
To update a shared dataset, run the command "datalad update --merge".
This command will query its origin for changes, and integrate the
changes into the dataset.

EOT

# save the changes

$ datalad save -m "add note about datalad update"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

PS: You might wonder whether there is also a sole **datalad update** command. Yes, there is – if you are a Git-user and know about branches and merging you can read the `Note for Git-users` below. However, a thorough explanation and demonstration will be in the next section.

> **G9.2 Update internals**
>
> **datalad update** is the DataLad equivalent of a **git fetch**, **datalad update --merge** is the DataLad equivalent of a **git pull**. Upon a simple **datalad update**, the remote information is available on a branch separate from the master branch – in most cases this

> will be `remotes/origin/master`. You can **git checkout** this branch or run **git diff** to explore the changes and identify potential merge conflicts.

9.5 Networking

To get a hang on the basics of sharing a dataset, you shared your `DataLad-101` dataset with your room mate on a common, local file system. Your lucky room mate now has your notes and can thus try to catch up to still pass the course. Moreover, though, he can also integrate all other notes or changes you make to your dataset, and stay up to date. This is because a DataLad dataset makes updating shared data a matter of a single **datalad update --merge** command.

But why does this need to be a one-way line? "I want to provide helpful information for you as well!", says your room mate. "How could you get any insightful notes that I make in my dataset, or maybe the results of our upcoming mid-term project? Its a bit unfair that I can get your work, but you can not get mine."

Consider, for example, that your room mate might have googled about DataLad a bit. On the [datalad homepage](91) he might have found very useful additional information, such as the ascii-cast on [dataset nesting](92). Because he found this very helpful in understanding dataset nesting concepts, he decided to download the `shell` script that was [used to generate this example](93) from GitHub, and saved it in the code/ directory.

He does it using the datalad command **datalad download-url** that you experienced in section *Create a dataset* (page 34) already: This command will download a file just as `wget`, but it can also take a commit message and will save the download right to the history of the dataset that you specify, while recording its origin as provenance information.

Navigate into your dataset copy in `mock_user/DataLad-101`, and run the following command

```
# navigate into the installed copy
$ cd ../mock_user/DataLad-101

# download the shell script and save it in your code/ directory
$ datalad download-url \
  -d . \
  -m "Include nesting demo from datalad website" \
  -O code/nested_repos.sh \
  https://raw.githubusercontent.com/datalad/datalad.org/
↪7e8e39b1f08d0a54ab521586f27ee918b4441d69/content/asciicast/seamless_nested_repos.sh
[INFO] Downloading 'https://raw.githubusercontent.com/datalad/datalad.org/
↪7e8e39b1f08d0a54ab521586f27ee918b4441d69/content/asciicast/seamless_nested_repos.sh'␣
↪into '/home/me/dl-101/mock_user/DataLad-101/code/nested_repos.sh'
download_url(ok): /home/me/dl-101/mock_user/DataLad-101/code/nested_repos.sh (file)
add(ok): code/nested_repos.sh (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
```

---

91 https://www.datalad.org/
92 https://www.datalad.org/for/git-users
93 https://raw.githubusercontent.com/datalad/datalad.org/7e8e39b1f08d0a54ab521586f27ee918b4441d69/content/asciicast/seamless_nested_repos.sh

Run a quick datalad status:

```
$ datalad status
nothing to save, working tree clean
```

Nice, the **datalad download-url** command saved this download right into the history, and **datalad status** does not report unsaved modifications! We'll show an excerpt of the last commit here:

```
$ git log -n 1 -p
commit e75e2dff6d91d9273a57d7e6e139df3229617576
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:22:16 2021 +0200

    Include nesting demo from datalad website

diff --git a/code/nested_repos.sh b/code/nested_repos.sh
new file mode 100644
index 0000000..f84c817
--- /dev/null
+++ b/code/nested_repos.sh
@@ -0,0 +1,59 @@
```

Suddenly, your room mate has a file change that you do not have. His dataset evolved.

So how do we link back from the copy of the dataset to its origin, such that your room mate's changes can be included in your dataset? How do we let the original dataset "know" about this copy your room mate has? Do we need to install the installed dataset of our room mate as a copy again?

No, luckily, it's simpler and less convoluted. What we have to do is to *register* a datalad SIBLING: A reference to our room mate's dataset in our own, original dataset.

> **G9.3 Remote siblings**
>
> Git repositories can configure clones of a dataset as *remotes* in order to fetch, pull, or push from and to them. A **datalad sibling** is the equivalent of a git clone that is configured as a remote.

Let's see how this is done.

First of all, navigate back into the original dataset. In the original dataset, "add" a "sibling" by using the **datalad siblings** command (datalad-siblings manual). The command takes the base command, **datalad siblings**, an action, in this case add, a path to the root of the dataset -d ., a name for the sibling, -s/--name roommate, and a URL or path to the sibling, --url ../mock_user/DataLad-101. This registers your room mate's DataLad-101 as a "sibling" (we will call it "roommate") to your own DataLad-101 dataset.

```
$ cd ../../DataLad-101
# add a sibling
$ datalad siblings add -d . \
  --name roommate --url ../mock_user/DataLad-101
.: roommate(+) [../mock_user/DataLad-101 (git)]
```

There are a few confusing parts about this command: For one, do not be surprised about the --url argument – it's called "URL" but it can be a path as well. Also, do not forget to give a

name to your dataset's sibling. Without the `-s/ --name` argument the command will fail. The reason behind this is that the default name of a sibling if no name is given will be the host name of the specified URL, but as you provide a path and not a URL, there is no host name to take as a default.

As you can see in the command output, the addition of a sibling succeeded: `roommate(+)[.` `./mock_user/DataLad-101]` means that your room mate's dataset is now known to your own dataset as "roommate"

```
$ datalad siblings
.: here(+) [git]
.: roommate(+) [../mock_user/DataLad-101 (git)]
```

This command will list all known siblings of the dataset. You can see it in the resulting list with the name "roommate" you have given to it.

> **M9.3 What if I mistyped the name or want to remove the sibling?**
>
> You can remove a sibling using `datalad siblings remove -s roommate`

The fact that the `DataLad-101` dataset now has a sibling means that we can also **datalad update** this repository. Awesome!

Your room mate previously ran a **datalad update --merge** in the section *Stay up to date* (page 103). This got him changes *he knew you made* into a dataset that *he so far did not change*. This meant that nothing unexpected would happen with the **datalad update --merge**.

But consider the current case: Your room mate made changes to his dataset, but you do not necessarily know which. You also made changes to your dataset in the meantime, and added a note on **datalad update**. How would you know that his changes and your changes are not in conflict with each other?

This scenario is where a plain **datalad update** becomes useful. If you run a plain **datalad update**, DataLad will query the sibling for changes, and store those changes in a safe place in your own dataset, *but it will not yet integrate them into your dataset*. This gives you a chance to see whether you actually want to have the changes your room mate made.

Let's see how it's done. First, run a plain **datalad update** without the `--merge` option.

```
$ datalad update -s roommate
[INFO] Fetching updates for Dataset(/home/me/dl-101/DataLad-101)
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
update(ok): . (dataset)
```

Note that we supplied the sibling's name with the `-s/--name` option. This is good practice, and allows you to be precise in where you want to get updates from. It would have worked without the specification (just as a bare **datalad update --merge** worked for your room mate), because there is only one other known location, though.

This plain **datalad update** informs you that it "fetched" updates from the dataset. The changes however, are not yet visible – the script that he added is not yet in your code/ directory:

```
$ ls code/
list_titles.sh
```

So where is the file? It is in a different *branch* of your dataset.

If you do not use GIT, the concept of a BRANCH can be a big source of confusion. There will be sections later in this book that will elaborate a bit more what branches are, and how to work with them, but for now envision a branch just like a bunch of drawers on your desk. The paperwork that you have in front of you right on your desk is your dataset as you currently see it. These drawers instead hold documents that you are in principle working on, just not now – maybe different versions of paperwork you currently have in front of you, or maybe other files than the ones currently in front of you on your desk.

Imagine that a `datalad update` created a small drawer, placed all of the changed or added files from the sibling inside, and put it on your desk. You can now take a look into that drawer to see whether you want to have the changes right in front of you.

The drawer is a branch, and it is usually called `remotes/origin/master`. To look inside of it you can **git checkout BRANCHNAME**, or you can do a `diff` between the branch (your drawer) and the dataset as it is currently in front of you (your desk). We will do the latter, and leave the former for a different lecture:

> **W9.1 Please use datalad diff --from master --to remotes/roommate/master**
>
> Please use the following command instead:
>
> ```
> datalad diff --from master --to remotes/roommate/master
> ```
>
> This syntax specifies the MASTER BRANCH as a starting point for the comparison instead of the current `adjusted/master(unlocked)` branch.

```
$ datalad diff --to remotes/roommate/master
    added: code/nested_repos.sh (file)
 modified: notes.txt (file)
```

This shows us that there is an additional file, and it also shows us that there is a difference in `notes.txt`! Let's ask **git diff** to show us what the differences in detail (note that it is a shortened excerpt, cut in the middle to reduce its length):

> **W9.2 Please use git diff master..remotes/roommate/master**
>
> Please use the following command instead:
>
> ```
> git diff master..remotes/roommate/master
> ```
>
> This is GITs syntax for specifying a comparison between two BRANCHes.

```
$ git diff remotes/roommate/master
diff --git a/code/nested_repos.sh b/code/nested_repos.sh
deleted file mode 100644
index f84c817..0000000
--- a/code/nested_repos.sh
+++ /dev/null
@@ -1,59 +0,0 @@
-#!/bin/bash
-# This script was converted using cast2script from:
-# docs/casts/seamless_nested_repos.sh
-set -e -u
-export GIT_PAGER=cat
```

(continues on next page)

```
-
-# DataLad provides seamless management of nested Git repositories...
-
-# Let's create a dataset
-datalad create demo
-cd demo
diff --git a/notes.txt b/notes.txt
index 655be7d..3bf3281 100644
--- a/notes.txt
+++ b/notes.txt
@@ -59,3 +59,7 @@ The command "git annex whereis PATH" lists the repositories that have
 the file content of an annexed file. When using "datalad get" to
 retrieve file content, those repositories will be queried.

+To update a shared dataset, run the command "datalad update --merge".
+This command will query its origin for changes, and integrate the
+changes into the dataset.
+
```

Let's digress into what is shown here. We are comparing the current state of your dataset against the current state of your room mate's dataset. Everything marked with a - is a change that your room mate has, but not you: This is the script that he downloaded!

Everything that is marked with a + is a change that you have, but not your room mate: It is the additional note on **datalad update** you made in your own dataset in the previous section.

Cool! So now that you know what the changes are that your room mate made, you can safely **datalad update --merge** them to integrate them into your dataset. In technical terms you will "*merge the branch remotes/roommate/master into master*". But the details of this will be stated in a standalone section later.

Note that the fact that your room mate does not have the note on **datalad update** does not influence your note. It will not get deleted by the merge. You do not set your dataset to the state of your room mate's dataset, but you incorporate all changes he made – which is only the addition of the script.

```
$ datalad update --merge -s roommate
[INFO] Fetching updates for Dataset(/home/me/dl-101/DataLad-101)
merge(ok): . (dataset) [Merged roommate/master]
update(ok): . (dataset)
action summary:
  merge (ok: 1)
  update (ok: 1)
```

The exciting question is now whether your room mate's change is now also part of your own dataset. Let's list the contents of the code/ directory and also peek into the history:

```
$ ls code/
list_titles.sh
nested_repos.sh
```

```
$ git log --oneline
67a7a5e Merge remote-tracking branch 'roommate/master'
e75e2df Include nesting demo from datalad website
4069dac add note about datalad update
```

```
401df53 add note on git annex whereis
dfe8e39 add note about cloning from paths and recursive datalad get
```

Wohoo! Here it is: The script now also exists in your own dataset. You can see the commit that your room mate made when he saved the script, and you can also see a commit that records how you merged your room mate's dataset changes into your own dataset. The commit message of this latter commit for now might contain many words yet unknown to you if you do not use Git, but a later section will get into the details of what the meaning of "MERGE", "BRANCH", "refs" or "MASTER" is.

For now, you're happy to have the changes your room mate made available. This is how it should be! You helped him, and he helps you. Awesome! There actually is a wonderful word for it: *Collaboration*. Thus, without noticing, you have successfully collaborated for the first time using DataLad datasets.

Create a note about this, and save it.

```
$ cat << EOT >> notes.txt
To update from a dataset with a shared history, you need to add this
dataset as a sibling to your dataset. "Adding a sibling" means
providing DataLad with info about the location of a dataset, and a
name for it.
Afterwards, a "datalad update --merge -s name" will integrate the
changes made to the sibling into the dataset. A safe step in between
is to do a "datalad update -s name" and checkout the changes with
"git/datalad diff" to remotes/origin/master

EOT
$ datalad save -m "Add note on adding siblings"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

9.6 Summary

Together with your room mate you have just discovered how to share, update, and collaborate on a DataLad dataset on a shared file system. Thus, you have glimpsed into the principles and advantages of sharing a dataset with a simple example.

- To obtain a dataset, one can also use **datalad clone** with a path. Potential subdatasets will not be installed right away. As they are registered in the superdataset, you can do **datalad get -n/--no-data**, or specify the -r/--recursive (datalad get -n -r <subds>) with a decent -R/--recursion-limit choice to install them afterwards.

- The configuration of the original dataset determines which types of files will have their content available right after the installation of the dataset, and which types of files need to be retrieved via **datalad get**: Any file content stored in GIT will be available right away, while all file content that is annexed only has small metadata about its availability attached to it. The original DataLad-101 dataset used the text2git configuration template to store text files such as notes.txt and code/list_titles.sh in Git – these files' content is therefore available right after installation.

- Annexed content can be retrieved via **datalad get** from the file content sources.

- **git annex whereis PATH** will list all locations known to contain file content for a particular file. This location is where GIT-ANNEX will attempt to retrieve file content from, and it is described with the --description provided during a **datalad create**. It is a very helpful command to find out where file content resides, and how many locations with copies exist.

- A shared copy of a dataset includes the datasets history. If well made, **datalad run** commands can then easily be rerun.

- Because an installed dataset knows its origin – the place it was originally installed from – it can be kept up-to-date with the **datalad update** command. This command will query the origin of the dataset for updates, and a **datalad update --merge** will integrate these changes into the dataset copy.

- Thus, using DataLad, data can be easily shared and kept up to date with only two commands: **datalad clone** and **datalad update**.

- By configuring a dataset as a SIBLING, collaboration becomes easy.

- To avoid integrating conflicting modifications of a sibling dataset into your own dataset, a **datalad update -s SIBLINGNAME** will "fetch" modifications and store them on a different BRANCH of your dataset. The commands **datalad diff** and **git diff** can subsequently help to find out what changes have been made in the sibling.

Now what I can do with that?

Most importantly, you have experienced the first way of sharing and updating a dataset. The example here may strike you as too simplistic, but in later parts of the book you will see examples in which datasets are shared on the same file system in surprisingly useful ways.

Simultaneously, you have observed dataset properties you already knew (for example how annexed files need to be retrieved via **datalad get**), but you have also seen novel aspects of a dataset – for example that subdatasets are not automatically installed by default, how **git annex whereis** can help you find out where file content might be stored, how useful commands that capture provenance about the origin or creation of files (such as **datalad run** or **datalad download-url**) are, or how a shared dataset can be updated to reflect changes that were made to the original dataset.

Also, you have successfully demonstrated a large number of DataLad dataset principles to your room mate: How content stored in Git is present right away and how annexed content first needs to be retrieved, how easy a **datalad rerun** is if the original **datalad run** command was well specified, how a datasets history is shared and not only its data.

Lastly, with the configuration of a sibling, you have experienced one way to collaborate in a dataset, and with **datalad update --merge** and **datalad update**, you also glimpsed into more advances aspects of Git, namely the concept of a branch.

Therefore, these last few sections have hopefully been a good review of what you already knew, but also a big knowledge gain, and cause joyful anticipation of collaboration in a real-world setting of one of your own use cases.

TUNING DATASETS TO YOUR NEEDS

10.1 DIY configurations

Back in section *Data safety* (page 83), you already learned that there are dataset configurations, and that these configurations can be modified, for example with the `-c text2git` option. This option applies a configuration template to store text files in GIT instead of GIT-ANNEX, and thereby modifies the DataLad dataset's default configuration to store every file in git-annex.

The lecture today focuses entirely on the topic of configurations, and aims to equip everyone with the basics to configure their general and dataset specific setup to their needs. This is not only a handy way to tune a dataset to one's wishes, but also helpful to understand potential differences in command execution and file handling between two users, computers, or datasets.

"First of all, when we talk about configurations, we have to differentiate between different scopes of configuration, and different tools the configuration belongs or applies to", our lecturer starts. "In DataLad datasets, different tools can have a configuration: GIT, GIT-ANNEX, and DataLad itself. Because these tools are all combined by DataLad to help you manage your data, it is important to understand how the configuration of one software is used by or influences a second tool, or the overall dataset performance."

"Oh crap, one of these theoretical lectures again" mourns a student from the row behind you. Personally, you'd also be much more excited about any hands-on lecture filled with commands. But the recent lecture about GIT-ANNEX and the OBJECT-TREE was surprisingly captivating, so you're actually looking forward to today. "Shht! I want to hear this!", you shush him with a wink.

"We will start by looking into the very first configuration you did, already before the course started: The *global* Git configuration." the lecturer says.

At one point in time, you likely followed instructions such as in *Installation and configuration* (page 10) and configured your *Git identity* with the commands:

```
git config --global --add user.name "Elena Piscopia"
git config --global --add user.email elena@example.net
```

"What the above commands do is very simple: They search for a specific configuration file, and set the variables specified in the command – in this case user name and user email address – to the values provided with the command." she explains.

"This general procedure, specifying a value for a configuration variable in a configuration file, is how you can configure the different tools to your needs. The configuration, therefore, is really easy. Even if you are only used to ticking boxes in the settings tab of a software tool so far, it's intuitive to understand how a configuration file in principle works and also how to use it. The only piece of information you will need are the necessary files, or the command that writes to them, and the available options for configuration, that's it. And what's really cool is that all tools we'll be looking at – Git, git-annex, and DataLad – can be configured using the `git config` command[95]. Therefore, once you understand the syntax of this command, you already know half of what's relevant. The other half is understanding what you're doing. Now then, let's learn *how* to configure settings, but also *understand* what we're doing with these configurations."

"This seems easy enough", you think. Let's see what types of configurations there are.

Git config files

The user name and email configuration is a *user-specific* configuration (called *global* configuration by Git), and therefore applies to your user account. Wherever on your computer *you* run a Git, git-annex, or DataLad command, this global configuration will associate the name and email address you supplied in the `git config` commands above with this action. For example, whenever you datalad save, the information in this file is used for the history entry about commit author and email.

Apart from *global* Git configurations, there are also *system-wide*[96]and *repository* configurations. Each of these configurations resides in its own file. The global configuration is stored in a file called `.gitconfig` in your home directory. Among your name and email address, this file can

---

[95] As an alternative to a git config command, you could also run configuration templates or procedures (see *Configurations to go* (page 126)) that apply predefined configurations or in some cases even add the information to the configuration file by hand and save it using an editor of your choice.

[96] The third scope of a Git configuration are the system wide configurations. These are stored (if they exist) in /etc/gitconfig and contain settings that would apply to every user on the computer you are using. These configurations are not relevant for DataLad-101, and we will thus skip them. You can read more about Git's configurations and different files here[97].

[97] https://git-scm.com/docs/git-config

store general per-user configurations, such as a default editor[98], or highlighting options.

The *repository-specific* configurations apply to each individual repository. Their scope is more limited than the *global* configuration (namely to a single repository), but it can overrule global configurations: The more specific the scope of a configuration file is, the more important it is, and the variables in the more specific configuration will take precedence over variables in less specific configuration files. One could for example have VIM configured to be the default editor on a global scope, but could overrule this by setting the editor to nano in a given repository. For this reason, the repository-specific configuration does not reside in a file in your home directory, but in `.git/config` within every Git repository (and thus DataLad dataset).

Thus, there are three different scopes of Git configuration, and each is defined in a `config` file in a different location. The configurations will determine how Git behaves. In principle, all of these files can configure the same variables differently, but more specific scopes take precedence over broader scopes. Conveniently, not only can DataLad and git-annex be configured with the same command as Git, but in many cases they will also use exactly the same files as Git for their own configurations.

Let's find out how the repository-specific configuration file in the `DataLad-101` superdataset looks like:

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[annex]
        uuid = 142a8659-5566-466d-af37-5958d071a995
        version = 8
        backends = MD5E
[filter "annex"]
        smudge = git-annex smudge -- %f
        clean = git-annex smudge --clean -- %f
[submodule "recordings/longnow"]
        active = true
        url = https://github.com/datalad-datasets/longnow-podcasts.git
[remote "roommate"]
        url = ../mock_user/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
        annex-uuid = dfa64995-4862-40bf-b5c7-518db36f3f4d
        annex-ignore = false
```

---

[98] If your default editor is VIM and you do not like this, now can be the time to change it! Chose either of two options:

  1) Open up the file with an editor for your choice (e.g., nano[99]), and either paste the following configuration or edit it if it already exists:

```
[core]
    editor = nano
```

  2) Run the following command, but exchange nano with an editor of your choice:

```
$ git config --global --add core.editor "nano"
```

[99] https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/

This file consists of so called "sections" with the section names in square brackets (e.g., core). Occasionally, a section can have subsections: This is indicated by subsection names in quotation marks after the section name. For example, roommate is a subsection of the section remote. Within each section, variable = value pairs specify configurations for the given (sub)section.

The first section is called core – as the name suggests, this configures core Git functionality. There are many more[94] configurations than the ones in this config file, but they are related to Git, and less related or important to the configuration of a DataLad dataset. We will use this section to showcase the anatomy of the **git config** command. If for example you would want to specifically configure NANO to be the default editor in this dataset, you can do it like this:

```
$ git config --local --add core.editor "nano"
```

The command consists of the base command **git config**, a specification of the scope of the configuration with the --local flag, a name specification consisting of section and key with the notation section.variable (here: core.editor), and finally the value specification "nano".

Let's see what has changed:

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        editor = nano
[annex]
        uuid = 142a8659-5566-466d-af37-5958d071a995
        version = 8
        backends = MD5E
[filter "annex"]
        smudge = git-annex smudge -- %f
        clean = git-annex smudge --clean -- %f
[submodule "recordings/longnow"]
        active = true
        url = https://github.com/datalad-datasets/longnow-podcasts.git
[remote "roommate"]
        url = ../mock_user/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
        annex-uuid = dfa64995-4862-40bf-b5c7-518db36f3f4d
        annex-ignore = false
```

With this additional line in your repositories Git configuration, nano will be used as a default editor regardless of the configuration in your global or system-wide configuration. Note that the flag --local applies the configuration to your repository's .git/config file, whereas --global would apply it as a user specific configuration, and --system as a system-wide configuration. If you would want to change this existing line in your .git/config file, you would replace --add with --replace-all such as in:

```
git config --local --replace-all core.editor "vim"
```

to configure VIM to be your default editor. Note that while being a good toy example, it is not a common thing to configure repository-specific editors.

This example demonstrated the structure of a **git config** command. By specifying the name

---

[94] https://git-scm.com/docs/git-config#Documentation/git-config.txt-corefileMode

option with `section.variable` (or `section.subsection.variable` if there is a subsection), and a value, one can configure Git, git-annex, and DataLad. *Most* of these configurations will be written to a `config` file of Git, depending on the scope (local, global, system-wide) specified in the command.

> 💡 **M10.1 If things go wrong during Git config**
>
> If something goes wrong during the **git config** command, for example you end up having two keys of the same name because you added a key instead of replacing an existing one, you can use the `--unset` option to remove the line. Alternatively, you can also open the config file in an editor and remove or change sections by hand.

The only information you need, therefore, is the name of a section and variable to configure, and the value you want to specify. But in many cases it is also useful to find out which configurations are already set in which way and where. For this, the **git config --list --show-origin** is useful. It will display all configurations and their location:

```
$ git config --list --show-origin
file:/home/bob/.gitconfig    user.name=Bob McBobface
file:/home/bob/.gitconfig    user.email=bob@mcbobface.com
file:/home/bob/.gitconfig    core.editor=vim
file:/home/bob/.gitconfig    annex.security.allowed-url-schemes=http https file
file:.git/config    core.repositoryformatversion=0
file:.git/config    core.filemode=true
file:.git/config    core.bare=false
file:.git/config    core.logallrefupdates=true
file:.git/config    annex.uuid=1f83595e-bcba-4226-aa2c-6f0153eb3c54
file:.git/config    annex.version=5
file:.git/config    annex.backends=MD5E
file:.git/config    submodule.recordings/longnow.url=https://github.com/datalad-datasets/
↪longnow-podcasts.git
file:.git/config    submodule.recordings/longnow.active=true
file:.git/config    remote.roommate.url=../mock_user/onemoredir/DataLad-101
file:.git/config    remote.roommate.fetch=+refs/heads/*:refs/remotes/roommate/*
file:.git/config    remote.roommate.annex-uuid=a5ae24de-1533-4b09-98b9-cd9ba6bf303c
file:.git/config    remote.roommate.annex-ignore=false
file:.git/config    submodule.longnow.url=https://github.com/datalad-datasets/longnow-
↪podcasts.git
file:.git/config    submodule.longnow.active=true
```

This example shows some configurations in the global `.gitconfig` file, and the configurations within `DataLad-101/.git/config`. The command is very handy to display all configurations at once to identify configuration problems, find the right configuration file to make a change to, or simply remind oneself of the existing configurations, and it is a useful helper to keep in the back of your head.

At this point you may feel like many of these configurations or the configuration file inside of `DataLad-101` do not appear to be intuitively understandable enough to confidently apply changes to them, or identify necessary changes. And indeed, most of the sections and variables or values in there are irrelevant for understanding the book, your dataset, or DataLad, and can just be left as they are. The previous section merely served to de-mystify the **git config** command and the configuration files. Nevertheless, it might be helpful to get an overview about the meaning of the remaining sections in that file, and the *that disects this config file further* (page 131) can give you a glimpse of this.

`.git/config` versus other (configuration) files

One crucial aspect distinguishes the `.git/config` file from many other files in your dataset: Even though it is part of your dataset, it won't be shared together with the dataset. The reason for this is that this file is not version controlled, as it lies within the `.git` directory. Repository-specific configurations within your `.git/config` file are thus not written to history. Any local configuration in `.git/config` applies to the dataset, but it does not *stick* to the dataset. One can have the misconception that because the configurations were made *in* the dataset, these configurations will also be shared together with the dataset. `.git/config`, however, behaves just as your global or system-wide configurations. These configurations are in effect on a system, or for a user, or for a dataset, but are not shared. A **datalad clone** command of someone's dataset will not get you their editor configuration, should they have included one in their config file. Instead, upon a **datalad clone**, a new config file will be created.

This means, however, that configurations that should "stick" to a dataset[101] need to be defined in different files – files that are version controlled. The next section will talk about them.

10.2 More on DIY configurations

As the last section already suggest, within a Git repository, `.git/config` is not the only configuration file. There are also `.gitmodules` and `.gitattributes`, and in DataLad datasets there also is a `.datalad/config` file.

All of these files store configurations, but have an important difference: They are version controlled, and upon sharing a dataset these configurations will be shared as well. An example for a shared configuration is the one that the `text2git` configuration template applied: In the shared copy of your dataset, text files are also saved with Git, and not git-annex (see section *Networking* (page 105)). The configuration responsible for this behavior is in a `.gitattributes` file, and we'll start this section by looking into it.

`.gitattributes`

This file lies right in the root of your superdataset:

```
$ cat .gitattributes

* annex.backend=MD5E
**/.git* annex.largefiles=nothing
* annex.largefiles=((mimeencoding=binary)and(largerthan=0))
```

This looks neither spectacular nor pretty. Also, it does not follow the `section-option-value` organization of the `.git/config` file anymore. Instead, there are three lines, and all of these seem to have something to do with the configuration of git-annex. There even is one key word that you recognize: MD5E. If you have read the hidden section in *Data integrity* (page 85) you will recognize it as a reference to the type of key used by git-annex to identify and store file content in the object-tree. The first row, `* annex.backend=MD5E`, therefore translates to

---

[101] Please note that not all configurations can be written to files other than `.git/config`. Some of the files introduced in the next section will not be queried by Git, and in principle, it is a good thing that one can not share arbitrary configurations together with a dataset, as this could be a potential security threat. In those cases where you need dataset clones to inherit certain non-sticky configurations, it is advised to write a custom procedure and distribute it together with the dataset. The next two sections contain concrete usecases and tutorials.

"Everything in this directory should be hashed with a MD5E hash function". But what is the rest? We'll start with the last row:

```
* annex.largefiles=((mimeencoding=binary)and(largerthan=0))
```

Uhhh, cryptic. The lecturer explains:

"git-annex will *annex*, that is, *store in the object-tree*, anything it considers to be a "large file". By default, anything in your dataset would be a "large file", that means anything would be annexed. However, in section *Data integrity* (page 85) I already mentioned that exceptions to this behavior can be defined based on

1. file size

2. and/or path/pattern, and thus for example file extensions, or names, or file types (e.g., text files, as with the `text2git` configuration template).

"In `.gitattributes`, you can define what a large file and what is not by simply telling git-annex by writing such rules."

What you can see in this `.gitattributes` file is a rule based on **file types**: With `(mimeencoding=binary))`[106], the `text2git` configuration template configured git-annex to regard all files of type "binary" as a large file. Thanks to this little line, your text files are not annexed, but stored directly in Git.

The patterns `*` and `**` are so-called "wildcards" used in GLOBBING. `*` matches any file or directory in the current directory, and `**` matches all files and directories in the current directory *and subdirectories*. In technical terms, `**` matches *recursively*. The third row therefore translates to "Do not annex anything that is a text file in this directory" for git-annex.

However, rules can be even simpler. The second row simply takes a complete directory (`.git`) and instructs git-annex to regard nothing in it as a "large file". The second row, `**/.git* annex.largefiles=nothing` therefore means that no `.git` repository in this directory or a subdirectory should be considered a "large file". This way, the `.git` repositories are protected from being annexed. If you had a single file (`myfile.pdf`) you would not want annexed, specifying a rule such as:

```
myfile.pdf annex.largefiles=nothing
```

will keep it stored in Git. To see an example of this, navigate into the longnow subdataset, and view this dataset's `.gitattributes` file:

```
$ cat recordings/longnow/.gitattributes
* annex.backend=MD5E
**/.git* annex.largefiles=nothing
README.md annex.largefiles=nothing
```

The relevant part is `README.md annex.largefiles=nothing`. This instructs git-annex to specifically not annex `README.md`.

---

[106] When opening any file on a UNIX system, the file does not need to have a file extension (such as `.txt`, `.pdf`, `.jpg`) for the operating system to know how to open or use this file (in contrast to Windows, which does not know how to open a file without an extension). To do this, Unix systems rely on a file's MIME type – an information about a file's content. A `.txt` file for example has MIME type `text/plain` as does a bash script (`.sh`), a Python script has MIME type `text/x-python`, a `.jpg` file is `image/jpg`, and a `.pdf` file has MIME type `application/pdf`. You can find out the MIME type of a file by running:

```
$ file --mime-type path/to/file
```

Lastly, if you wanted to configure a rule based on **size**, you could add a row such as:

```
** annex.largefiles(largerthan=20kb)
```

to store only files exceeding 20KB in size in git-annex[107].

As you may have noticed, unlike `.git/config` files, there can be multiple `.gitattributes` files within a dataset. So far, you have seen one in the root of the superdataset, and in the root of the `longnow` subdataset. In principle, you can add one to every directory-level of your dataset. For example, there is another `.gitattributes` file within the `.datalad` directory:

```
$ cat .datalad/.gitattributes

config annex.largefiles=nothing
metadata/aggregate* annex.largefiles=nothing
metadata/objects/** annex.largefiles=(anything)
```

As with Git configuration files, more specific or lower-level configurations take precedence over more general or higher-level configurations. Specifications in a subdirectory can therefore override specifications made in the `.gitattributes` file of the parent directory.

In summary, the `.gitattributes` files will give you the possibility to configure what should be annexed and what should not be annexed up to individual file level. This can be very handy, and allows you to tune your dataset to your custom needs. For example, files you will often edit by hand could be stored in Git if they are not too large to ease modifying them[108]. Once you know the basics of this type of configuration syntax, writing your own rules is easy. For more tips on how configure git-annex's content management in `.gitattributes`, take a look at this[102] page of the git-annex documentation. Later however you will see preconfigured DataLad *procedures* such as `text2git` that can apply useful configurations for you, just as `text2git` added the last line in the root `.gitattributes` file.

`.gitmodules`

On last configuration file that Git creates is the `.gitmodules` file. There is one right in the root of your dataset:

```
$ cat .gitmodules
[submodule "recordings/longnow"]
        path = recordings/longnow
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
        datalad-url = https://github.com/datalad-datasets/longnow-podcasts.git
```

---

[107] Specifying annex.largefiles in your .gitattributes file will make the configuration "portable" – shared copies of your dataset will retain these configurations. You could however also set largefiles rules in your `.git/config` file. Rules specified in there take precedence over rules in `.gitattributes`. You can set them using the **git config** command:

```
$ git config annex.largefiles 'largerthan=100kb and not (include=*.c or include=*.h)'
```

The above command annexes files larger than 100KB, and will never annex files with a `.c` or `.h` extension.

[108] Should you ever need to, this file is also where one would change the git-annex backend in order to store new files with a new backend. Switching the backend of *all* files (new as well as existing ones) requires the **git annex migrate** command (see the documentation[109] for more information on this command).

[109] https://git-annex.branchable.com/git-annex-migrate/

[102] https://git-annex.branchable.com/tips/largefiles/

Based on these contents, you might have already guessed what this file stores. `.gitmodules` is a configuration file that stores the mapping between your own dataset and any subdatasets you have installed in it. There will be an entry for each submodule (subdataset) in your dataset. The name *submodule* is Git terminology, and describes a Git repository inside of another Git repository, i.e., the super- and subdataset principles. Upon sharing your dataset, the information about subdatasets and where to retrieve them from is stored and shared with this file.

Section *Looking without touching* (page 92) already mentioned one additional configuration option in a footnote: The `datalad-recursiveinstall` key. This key is defined on a per sub-dataset basis, and if set to "skip", the given subdataset will not be recursively installed unless it is explicitly specified as a path to **datalad get [-n/--no-data] -r**. If you are a maintainer of a superdataset with monstrous amounts of subdatasets, you can set this option and share it together with the dataset to prevent an accidental, large recursive installation in particularly deeply nested subdatasets. Below is a minimally functional example on how to apply the configuration and how it works:

Let's create a dataset hierarchy to work with (note that we concatenate multiple commands into a single line using bash's "and" && operator):

```
# create a superdataset with two subdatasets
$ datalad create superds && cd superds && datalad create -d . subds1 && datalad create -d
↪. subds2
[INFO   ] Creating a new annex repo at /tmp/superds
create(ok): /tmp/superds (dataset)
[INFO   ] Creating a new annex repo at /tmp/superds/subds1
add(ok): subds1 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subds1 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
[INFO   ] Creating a new annex repo at /tmp/superds/subds2
add(ok): subds2 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subds2 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
```

Next, we create subdatasets in the subdatasets:

```
# create two subdatasets in subds1
$ cd subds1 && datalad create -d . subsubds1 && datalad create -d . subsubds2 && cd ../
[INFO   ] Creating a new annex repo at /tmp/superds/subds1/subsubds1
add(ok): subsubds1 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subsubds1 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
```

```
[INFO   ] Creating a new annex repo at /tmp/superds/subds1/subsubds2
add(ok): subsubds2 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subsubds2 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)


# create two subdatasets in subds2
$ cd subds2 && datalad create -d . subsubds1 && datalad create -d . subsubds2
[INFO   ] Creating a new annex repo at /tmp/superds/subds2/subsubds1
add(ok): subsubds1 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subsubds1 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
[INFO   ] Creating a new annex repo at /tmp/superds/subds2/subsubds2
add(ok): subsubds2 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): subsubds2 (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
```

Here is the directory structure:

```
$ cd ../ && tree
.
├── subds1
│   ├── subsubds1
│   └── subsubds2
└── subds2
    ├── subsubds1
    └── subsubds2


# save in the superdataset
datalad save -m "add a few sub and subsub datasets"
add(ok): subds1 (file)
add(ok): subds2 (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  save (ok: 1)
```

Now, we can apply the datalad-recursiveinstall configuration to skip recursive installations for subds1

```
$ git config -f .gitmodules --add submodule.subds1.datalad-recursiveinstall skip

# save this configuration
$ datalad save -m "prevent recursion into subds1, unless explicitly given as path"
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

If the dataset is cloned, and someone runs a recursive **datalad get**, the subdatasets of subds1 will not be installed, the subdatasets of subds2, however, will be.

```
# clone the dataset somewhere else
$ cd ../ && datalad clone superds clone_of_superds
[INFO   ] Cloning superds into '/tmp/clone_of_superds'
install(ok): /tmp/clone_of_superds (dataset)

# recursively get all contents (without data)
$ cd clone_of_superds && datalad get -n -r .
[INFO   ] Installing <Dataset path=/tmp/clone_of_superds> underneath /tmp/clone_of_
↪superds recursively
[INFO   ] Cloning /tmp/superds/subds2 into '/tmp/clone_of_superds/subds2'
get(ok): /tmp/clone_of_superds/subds2 (dataset)
[INFO   ] Cloning /tmp/superds/subds2/subsubds1 into '/tmp/clone_of_superds/subds2/
↪subsubds1'
get(ok): /tmp/clone_of_superds/subds2/subsubds1 (dataset)
[INFO   ] Cloning /tmp/superds/subds2/subsubds2 into '/tmp/clone_of_superds/subds2/
↪subsubds2'
get(ok): /tmp/clone_of_superds/subds2/subsubds2 (dataset)
action summary:
  get (ok: 3)

# only subsubds of subds2 are installed, not of subds1:
$ tree
.
├── subds1
└── subds2
    ├── subsubds1
    └── subsubds2

4 directories, 0 files
```

Nevertheless, if subds1 is provided with an explicit path, its subdataset subsubds will be cloned, essentially overriding the configuration:

```
$  datalad get -n -r subds1 && tree
[INFO   ] Cloning /tmp/superds/subds1 into '/tmp/clone_of_superds/subds1'
install(ok): /tmp/clone_of_superds/subds1 (dataset) [Installed subdataset in order to get↪
↪/tmp/clone_of_superds/subds1]
[INFO   ] Installing <Dataset path=/tmp/clone_of_superds> underneath /tmp/clone_of_
↪superds/subds1 recursively
.
├── subds1
│   ├── subsubds1
│   └── subsubds2
```

(continues on next page)

```
└── subds2
    ├── subsubds1
    └── subsubds2

6 directories, 0 files
```

`.datalad/config`

DataLad adds a repository-specific configuration file as well. It can be found in the `.datalad` directory, and just like `.gitattributes` and `.gitmodules` it is version controlled and is thus shared together with the dataset. One can configure many options[103], but currently, our `.datalad/config` file only stores a DATASET ID. This ID serves to identify a dataset as a unit, across its entire history and flavors. In a geeky way, this is your dataset's social security number: It will only exist one time on this planet.

```
$ cat .datalad/config
[datalad "dataset"]
        id = 8e04afb0-af85-4070-be29-858d30d85017
```

Note, though, that local configurations within a Git configuration file will take precedence over configurations that can be distributed with a dataset. Otherwise, dataset updates with **datalad update** (or, for Git-users, `git pull`) could suddenly and unintentionally alter local DataLad behavior that was specifically configured. Also, GIT and GIT-ANNEX will not query this file for configurations, so please store only sticky options that are specific to DataLad (i.e., under the `datalad.*` namespace) in it.

Writing to configuration files other than `.git/config`

"Didn't you say that knowing the `git config` command is already half of what I need to know?" you ask. "Now there are three other configuration files, and I do not know with which command I can write into these files."

"Excellent question", you hear in return, "but in reality, you **do** know: it's also the `git config` command. The only part of it you need to adjust is the `-f`, `--file` parameter. By default, the command writes to a Git config file. But it can write to a different file if you specify it appropriately. For example

```
git config --file=.gitmodules --replace-all submodule."name".url "new URL"
```

will update your submodule's URL. Keep in mind though that you would need to commit this change, as `.gitmodules` is version controlled".

Let's try this:

```
$ git config --file=.gitmodules --replace-all submodule."recordings/longnow".url
↪"git@github.com:datalad-datasets/longnow-podcasts.git"
```

This command will replace the submodule's https URL with an SSH URL. The latter is often used if someone has an *SSH key pair* and added the public key to their GitHub account (you

---

[103] http://docs.datalad.org/en/latest/generated/datalad.config.html

can read more about this here[104]). We will revert this change shortly, but use it to show the difference between a **git config** on a .git/config file and on a version controlled file:

```
$ datalad status
 modified: .gitmodules (file)
```

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 9bc9ee9..11273e1 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,5 +1,5 @@
 [submodule "recordings/longnow"]
        path = recordings/longnow
-       url = https://github.com/datalad-datasets/longnow-podcasts.git
+       url = git@github.com:datalad-datasets/longnow-podcasts.git
        datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
      datalad-url = https://github.com/datalad-datasets/longnow-podcasts.git
```

As these two commands show, the .gitmodules file is modified. The https URL has been deleted (note the -, and a SSH URL has been added. To keep these changes, we would need to **datalad save** them. However, as we want to stay with https URLs, we will just *checkout* this change – using a Git tool to undo an unstaged modification.

```
$ git checkout .gitmodules
$ datalad status
Updated 1 path from the index
nothing to save, working tree clean
```

Note, though, that the .gitattributes file can not be modified with a **git config** command. This is due to its different format that does not comply to the section.variable.value structure of all other configuration files. This file, therefore, has to be edited by hand, with an editor of your choice.

Environment variables

An ENVIRONMENT VARIABLE is a variable set up in your shell that affects the way the shell or certain software works – for example the environment variables HOME, PWD, or PATH. Configuration options that determine the behavior of Git, git-annex, and DataLad that could be defined in a configuration file can also be set (or overridden) by the associated environment variables of these configuration options. Many configuration items have associated environment variables. If this environment variable is set, it takes precedence over options set in configuration files, thus providing both an alternative way to define configurations as well as an override mechanism. For example, the user.name configuration of Git can be overridden by its associated environment variable, GIT_AUTHOR_NAME. Likewise, one can define the environment variable instead of setting the user.name configuration in a configuration file.

Git, git-annex, and DataLad have more environment variables than anyone would want to remember. Here[105] is a good overview on Git's most useful available environment variables for a start. All of DataLad's configuration options can be translated to their associated environment variables. Any environment variable with a name that starts with DATALAD_ will be available as the corresponding datalad. configuration variable, replacing any __ (two underscores)

---

[104] https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories
[105] https://git-scm.com/book/en/v2/Git-Internals-Environment-Variables

with a hyphen, then any _ (single underscore) with a dot, and finally converting all letters to lower case. The `datalad.log.level` configuration option thus is the environment variable `DATALAD_LOG_LEVEL`.

> **M10.3 Some more general information on environment variables**
>
> Names of environment variables are often all-uppercase. While the `$` is not part of the name of the environment variable, it is necessary to *refer* to the environment variable: To reference the value of the environment variable `HOME` for example you would need to use `echo $HOME` and not `echo HOME`. However, environment variables are set without a leading `$`. There are several ways to set an environment variable (note that there are no spaces before and after the = !), leading to different levels of availability of the variable:
> - `THEANSWER=42 <command>` makes the variable `THEANSWER` available for the process in `<command>`. For example, `DATALAD_LOG_LEVEL=debug datalad get <file>` will execute the **datalad get** command (and only this one) with the log level set to "debug".
> - `export THEANSWER=42` makes the variable `THEANSWER` available for other processes in the same session, but it will not be available to other shells.
> - `echo 'export THEANSWER=42' >> ~/.bashrc` will write the variable definition in the `.bashrc` file and thus available to all future shells of the user (i.e., this will make the variable permanent for the user)
>
> To list all of the configured environment variables, type `env` into your terminal.

Summary

This has been an intense lecture, you have to admit. One definite take-away from it has been that you now know a second reason why the hidden `.git` and `.datalad` directory contents and also the contents of `.gitmodules` and `.gitattributes` should not be carelessly tampered with – they contain all of the repositories configurations.

But you now also know how to modify these configurations with enough care and background knowledge such that nothing should go wrong once you want to work with and change them. You can use the **git config** command for Git configuration files on different scopes, and even the `.gitmodules` or `datalad/config` files. Of course you do not yet know all of the available configuration options. However, you already know some core Git configurations such as name, email, and editor. Even more important, you know how to configure git-annex's content management based on `largefile` rules, and you understand the majority of variables within `.gitmodules` or the sections in `.git/config`. Slowly, you realize with pride, you're more and more becoming a DataLad power-user.

Write a note about configurations in datasets into `notes.txt`.

```
$ cat << EOT >> notes.txt
Configurations for datasets exist on different levels (systemwide,
global, and local), and in different types of files (not version
controlled (git)config files, or version controlled .datalad/config,
.gitattributes, or gitmodules files), or environment variables.
With the exception of .gitattributes, all configuration files share a
common structure, and can be modified with the git config command, but
also with an editor by hand.

Depending on whether a configuration file is version controlled or
```

```
not, the configurations will be shared together with the dataset.
More specific configurations and not-shared configurations will always
take precedence over more global or hared configurations, and
environment variables take precedence over configurations in files.

The git config --list --show-origin command is a useful tool to give
an overview over existing configurations. Particularly important may
be the .gitattributes file, in which one can set rules for git-annex
about which files should be version-controlled with Git instead of
being annexed.

EOT
```

```
$ datalad save -m "add note on configurations and git config"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

10.3 Configurations to go

The past two sections should have given you a comprehensive overview on the different configuration options the tools Git, git-annex, and DataLad provide. They not only showed you a way to configure everything you may need to configure, but also gave explanations about what the configuration options actually mean.

But figuring out which configurations are useful and how to apply them are also not the easiest tasks. Therefore, some clever people decided to assist with these tasks, and created preconfigured *procedures* that process datasets in a particular way. These procedures can be shipped within DataLad or its extensions, lie on a system, or can be shared together with datasets.

One of such procedures is the text2git configuration. In order to learn about procedures in general, let's demystify what the text2git procedure exactly is: It is nothing more than a simple script that

- writes the relevant configuration (annex_largefiles = '((mimeencoding=binary)and(largerthan=0))', i.e., "Do not put anything that is a text file in the annex") to the .gitattributes file of a dataset, and

- saves this modification with the commit message "Instruct annex to add text files to Git".

This particular procedure lives in a script called cfg_text2git in the sourcecode of DataLad. The amount of code in this script is not large, and the relevant lines of code are highlighted:

```
import sys
import os.path as op

from datalad.distribution.dataset import require_dataset

ds = require_dataset(
    sys.argv[1],
    check_installed=True,
```

```
    purpose='configuration')

# the relevant configuration:
annex_largefiles = '((mimeencoding=binary)and(largerthan=0))'
attrs = ds.repo.get_gitattributes('*')
if not attrs.get('*', {}).get(
        'annex.largefiles', None) == annex_largefiles:
    ds.repo.set_gitattributes([
        ('*', {'annex.largefiles': annex_largefiles})])

git_attributes_file = op.join(ds.path, '.gitattributes')
ds.save(
    git_attributes_file,
    message="Instruct annex to add text files to Git",
)
```

Just like `cfg_text2git`, all DataLad procedures are executables (such as a script, or compiled code). In principle, they can be written in any language, and perform any task inside of a dataset. The `text2git` configuration for example applies a configuration for how git-annex treats different file types. Other procedures do not only modify `.gitattributes`, but can also populate a dataset with particular content, or automate routine tasks such as synchronizing dataset content with certain siblings. What makes them a particularly versatile and flexible tool is that anyone can write their own procedures. If a workflow is a standard in a team and needs to be applied often, turning it into a script can save time and effort. To learn how to do this, read the *with a tutorial on writing own procedures* (page 132). By pointing DataLad to the location the procedures reside in they can be applied, and by including them in a dataset they can even be shared. And even if the script is simple, it is very handy to have preconfigured procedures that can be run in a single command line call. In the case of `text2git`, all text files in a dataset will be stored in Git – this is a useful configuration that is applicable to a wide range of datasets. It is a shortcut that spares naive users the necessity to learn about the `.gitattributes` file when setting up a dataset.

To find out available procedures, the command **datalad run-procedure --discover** (datalad-run-procedure manual) is helpful. This command will make DataLad search the default location for procedures in a dataset, the source code of DataLad or installed DataLad extensions, and the default locations for procedures on the system for available procedures:

```
$ datalad run-procedure --discover
cfg_bids (/home/adina/env/handbook2/lib/python3.9/site-packages/datalad_neuroimaging/
→resources/procedures/cfg_bids.py) [python_script]
cfg_hirni (/home/adina/env/handbook2/lib/python3.9/site-packages/datalad_hirni/resources/
→procedures/cfg_hirni.py) [python_script]
cfg_metadatatypes (/home/adina/repos/datalad/datalad/resources/procedures/cfg_
→metadatatypes.py) [python_script]
cfg_text2git (/home/adina/repos/datalad/datalad/resources/procedures/cfg_text2git.py)
→[python_script]
cfg_yoda (/home/adina/repos/datalad/datalad/resources/procedures/cfg_yoda.py) [python_
→script]
```

The output shows that in this particular dataset, on the particular system the book is written on, there are at least three procedures available: `cfg_metadatatypes`, `cfg_text2git`, and `cfg_yoda`. It also lists where they are stored – in this case, they are all part of the source code of DataLad[110].

---

[110] In theory, because procedures can exist on different levels, and because anyone can create (and thus name) their own procedures, there can be name conflicts. The order of precedence in such cases is: user-level, system-level,

- cfg_yoda configures a dataset according to the yoda principles – the section *YODA: Best practices for data analyses in a dataset* (page 136) talks about this in detail.
- cfg_text2git configures text files to be stored in Git.
- cfg_metadatatypes lets users configure additional metadata types – more about this in a later section on DataLad's metadata handling.

Applying procedures

datalad run-procedure not only *discovers* but also *executes* procedures. If given the name of a procedure, this command will apply the procedure to the current dataset, or the dataset that is specified with the -d/--dataset flag:

```
datalad run-procedure [-d <PATH>] cfg_text2git
```

The typical workflow is to create a dataset and apply a procedure afterwards. However, some procedures shipped with DataLad or its extensions with a cfg_ prefix can also be applied right at the creation of a dataset with the -c/--cfg-proc <name> option in a **datalad create** command. This is a peculiarity of these procedures because, by convention, all of these procedures are written to not require arguments. The command structure looks like this:

```
datalad create -c text2git DataLad-101
```

Note that the cfg_ prefix of the procedures is omitted in these calls to keep it extra simple and short. The available procedures in this example (cfg_yoda, cfg_text2git) could thus be applied within a **datalad create** as

- datalad create -c yoda <DSname>
- datalad create -c text2git <DSname>

> **M10.4 Applying multiple procedures**
>
> If you want to apply several configurations at once, feel free to do so, for example like this:
>
> ```
> $ datalad create -c yoda -c text2git
> ```

> **M10.5 Applying procedures in subdatasets**
>
> Procedures can be applied in datasets on any level in the dataset hierarchy, i.e., also in subdatasets. Note, though, that a subdataset will show up as being modified in **datalad status** *in the superdataset* after applying a procedure. This is expected, and it would also be the case with any other modification (saved or not) in the subdataset, as the version of the subdataset that is tracked in the superdataset simply changed. A **datalad save** in the superdataset will make sure that the version of the subdataset gets updated in the superdataset. The section *More on Dataset nesting* (page 164) will elaborate on this general principle later in the handbook.

---

dataset, DataLad extension, DataLad, i.e., local procedures take precedence over those coming from "outside" via datasets or DataLad extensions. If procedures in a higher-level dataset and a subdataset have the same name, the procedure closer to the dataset run-procedure is operating on takes precedence.

As a general note, it can be useful to apply procedures early in the life of a dataset. Procedures such as `cfg_yoda` (explained in detail in section *YODA: Best practices for data analyses in a dataset* (page 136)), create files, change `.gitattributes`, or apply other configurations. If many other (possibly complex) configurations are already in place, or if files of the same name as the ones created by a procedure are already in existence, this can lead to unexpected problems or failures, especially for naive users. Applying `cfg_text2git` to a default dataset in which one has saved many text files already (as per default added to the annex) will not place the existing, saved files into Git – only those text files created *after* the configuration was applied.

Summing up, DataLad's **run-procedure** command is a handy tool with useful existing procedures but much flexibility for your own DIY procedure scripts. With the information of the last three sections you should be able to write and understand necessary configurations, but you can also rely on existing, preconfigured templates in the form of procedures, and even write and distribute your own.

Therefore, envision procedures as helper-tools that can minimize technical complexities in a dataset – users can concentrate on the actual task while the dataset is set-up, structured, processed, or configured automatically with the help of a procedure. Especially in the case of trainees and new users, applying procedures instead of doing relevant routines "by hand" can help to ease working with the dataset, as the use case *Student supervision in a research project* (page 407) showcases. Other than by users, procedures can also be triggered to automatically run after any command execution if a command results matches a specific requirement. If you are interested in finding out more about this, read on in section *DataLad's result hooks* (page 287).

Finally, make a note about running procedures inside of `notes.txt`:

```
$ cat << EOT >> notes.txt
It can be useful to use pre-configured procedures that can apply
configurations, create files or file hierarchies, or perform arbitrary
tasks in datasets. They can be shipped with DataLad, its extensions,
or datasets, and you can even write your own procedures and distribute
them.
The "datalad run-procedure" command is used to apply such a procedure
to a dataset. Procedures shipped with DataLad or its extensions
starting with a "cfg" prefix can also be applied at the creation of a
dataset with "datalad create -c <PROC-NAME> <PATH>" (omitting the
"cfg" prefix).

EOT
```

```
$ datalad save -m "add note on DataLad's procedures"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

10.4 Summary

This has been a substantial amount of information regarding various configuration types, methods, and files. After this lecture, you have greatly broadened your horizon about configurations of datasets:

- Configurations exist at different scopes and for different tools. Each of such configuration scopes exists in an individual file, on a *system-wide*, *global* (user-specific) or *local* (repository specific) level. In addition to Git's *local* scope in `.git/config`, DataLad introduces its own configurations within `.datalad/config` that apply to a specific dataset, but are committed and therefore distributed. More specialized scopes take precedence over more global scopes.

- Almost all configurations can be set with the **git config**. Its structure looks like this:

  ```
  git config --local/--global/--system --add/remove-all/--list section.[subsection.
  ↪]variable "value"
  ```

- The `.git/config` configuration file is not version controlled, other configuration files (`.gitmodules`, `.gitattributes`, `.datalad/config`) however are, and can be shared together with the dataset. Non-shared configurations will take precedence over shared configurations in a dataset clone.

- Other tools than Git can be configured with the **git config** command as well. If the configuration needs to be written to a file other than a `.git(/)config` file, supply a path to this file with the `-f/--file` flag in a **git config** command.

- The `.gitattributes` file is the only configuration file the **git config** can not write to, because it has a different layout. However, run-procedures or the user can write simple rules into it that determine which files are annexed and which are stored in Git.

- DataLad's run-procedures offer an easy and fast alternative to DIY configurations, structuring, or processing of the dataset, and offer means to share or ship configurations together with a dataset. They can be applied already at creation of a dataset with `datalad create -c <procedure>`, or executed later with a **datalad run-procedure** command.

Now what can I do with it?

Configurations are not a closed book for you anymore. What will probably be especially helpful is your new knowledge about `.gitattributes` and DataLad's `run-procedure` command that allow you to configure the behavior of git-annex in your dataset.

> **💡 M10.2 Disecting a Git config file further**
>
> Let's walk through the Git config file of `DataLad-101`: The second section of `.git/config` is a git-annex configuration. As mentioned above, git-annex will use the GIT CONFIG FILE for some of its configurations. For example, it lists the repository as a "version 8 repository", and gives the dataset its own git-annex UUID. While the "annex-uuid"[100] looks like yet another cryptic random string of characters, you have seen a UUID like this before: A **git annex whereis** displays information about where the annexed content in a dataset is with these UUIDs. This section also specifies the supported backends in this dataset. If you have read the hidden section in the section *Data integrity* (page 85) you will recognize the name "MD5E". This is the hash function used to generate the annexed files keys and thus paths in the object tree. All backends specified in this file (it can be a list) can be used to hash your files.
>
> You may recognize the third part of the configuration, the subsection `"recordings/longnow"` in the section `submodule`. Clearly, this is a reference to the `longnow` podcasts we cloned as a subdataset. The name *submodule* is Git terminology, and describes a Git repository inside of another Git repository – just like the super- and subdataset principles you discovered in the section *Dataset nesting* (page 53). When you clone a DataLad dataset as a subdataset, it gets *registered* in this file. For each subdataset, an individual submodule entry will store the information about the subdataset's `--source` or *origin* (the "url"). Thus, every subdataset (and sub-subdataset, and so forth) in your dataset will be listed in this file. If you want, go back to section *Install datasets* (page 46) to see that the "url" is the same URL we cloned the longnow dataset from, and go back to section *Looking without touching* (page 92) to remind yourself of how cloning a dataset with subdatasets looked and felt like.
>
> Another interesting part is the last section, "remote". Here we can find the SIBLING "roommate" we defined in *Networking* (page 105). The term REMOTE is Git-terminology and is used to describe other repositories or DataLad datasets that the repository knows about and tracks. This file, therefore, is where DataLad *registered* the sibling with **datalad siblings add**, and thanks to it you can collaborate with your room mate. Note the *path* given as a value to the `url` variable. If at any point either your superdataset or the remote moves on your file system, the association between the two datasets breaks – this can be fixed by adjusting this path, and a demonstration of this is in section *Miscellaneous file system operations* (page 224). *fetch* contains a specification which parts of the repository are updated – in this case everything (all of the branches). Lastly, the `annex-ignore = false` configuration allows git-annex to query the remote when it tries to retrieve data from annexed content.
>
> ---
> [100] A UUID is a universally unique identifier – a 128-bit number that unambiguously identifies information.

> 💡 **M10.6 Write your own procedures**
>
> Procedures can come with DataLad or its extensions, but anyone can write their own ones in addition, and deploy them on individual machines, or ship them within DataLad datasets. This allows to automate routine configurations or tasks in a dataset, or share configurations that would otherwise not "stick" to the dataset. Some general rules for creating a custom procedure are outlined below:
>
> - A procedure can be any executable. Executables must have the appropriate permissions and, in the case of a script, must contain an appropriate SHEBANG.
>   - If a procedure is not executable, but its filename ends with .sh, it is automatically executed via BASH.
> - Procedures can implement any argument handling, but must be capable of taking at least one positional argument (the absolute path to the dataset they shall operate on).
> - Custom procedures rely heavily on configurations in .datalad/config (or the associated environment variables). Within .datalad/config, each procedure should get an individual entry that contains at least a short "help" description on what the procedure does. Below is a minimal .datalad/config entry for a custom procedure:
>
> ```
> [datalad "procedures.<NAME>"]
>     help = This is a string to describe what the procedure does
> ```
>
> - By default, on GNU/Linux systems, DataLad will search for system-wide procedures (i.e., procedures on the *system* level) in /etc/xdg/datalad/procedures, for user procedures (i.e., procedures on the *global* level) in ~/.config/datalad/procedures, and for dataset procedures (i.e., the *local* level[111]) in .datalad/procedures relative to a dataset root. Note that .datalad/procedures does not exist by default, and the procedures directory needs to be created first.
>   - Alternatively to the default locations, DataLad can be pointed to the location of a procedure with a configuration in .datalad/config (or with the help of the associated ENVIRONMENT VARIABLEs). The appropriate configuration keys for .datalad/config are either datalad.locations.system-procedures (for changing the *system* default), datalad.locations.user-procedures (for changing the *global* default), or datalad.locations.dataset-procedures (for changing the *local* default). An example .datalad/config entry for the local scope is shown below.
>
> ```
> [datalad "locations"]
>     dataset-procedures = relative/path/from/dataset-root
> ```
>
> - By default, DataLad will call a procedure with a standard template defined by a format string:
>
> ```
> interpreter {script} {ds} {arguments}
> ```
>
> where arguments can be any additional command line arguments a script (procedure) takes or requires. This default format string can be customized within .datalad/config in datalad.procedures.<NAME>.call-format. An example .datalad/config entry with a changed call format string is shown below.
>
> ```
> [datalad "procedures.<NAME>"]
>     help = This is a string to describe what the procedure does
>     call-format = python {script} {ds} {somearg1} {somearg2}
> ```

- By convention, procedures should leave a dataset in a clean state.

Therefore, in order to create a custom procedure, an executable script in the appropriate location is fine. Placing a script `myprocedure` into `.datalad/procedures` will allow running `datalad run-procedure myprocedure` in your dataset, and because it is part of the dataset it will also allow distributing the procedure. Below is a toy-example for a custom procedure:

```
$ datalad create somedataset; cd somedataset
[INFO] Creating a new annex repo at /home/me/procs/somedataset
[INFO] scanning for unlocked files (this may take some time)
create(ok): /home/me/procs/somedataset (dataset)
```

```
$ mkdir .datalad/procedures
$ cat << EOT > .datalad/procedures/example.py
"""A simple procedure to create a file 'example' and store
it in Git, and a file 'example2' and annex it. The contents
of 'example' must be defined with a positional argument."""

import sys
import os.path as op
from datalad.distribution.dataset import require_dataset
from datalad.utils import create_tree

ds = require_dataset(
    sys.argv[1],
    check_installed=True,
    purpose='showcase an example procedure')

# this is the content for file "example"
content = """\
This file was created by a custom procedure! Neat, huh?
"""

# create a directory structure template. Write
tmpl = {
    'somedir': {
        'example': content,
    },
    'example2': sys.argv[2] if sys.argv[2] else "got no input"
}

# actually create the structure in the dataset
create_tree(ds.path, tmpl)

# rule to store 'example' Git
ds.repo.set_gitattributes([('example', {'annex.largefiles': 'nothing'})])

# save the dataset modifications
ds.save(message="Apply custom procedure")

EOT
```

```
$ datalad save -m "add custom procedure"
add(ok): .datalad/procedures/example.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

At this point, the dataset contains the custom procedure example. This is how it can be executed and what it does:

```
$ datalad run-procedure example "this text will be in the file 'example2'"
[INFO] Running procedure example
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
```

```
#the directory structure has been created
$ tree
.
├── example2 -> .git/annex/objects/G6/zw/MD5E-s40--2ed1bce0db9f376c277a1ba6418f3ddd/
↪MD5E-s40--2ed1bce0db9f376c277a1ba6418f3ddd
└── somedir
    └── example

1 directory, 2 files
```

```
#lets check out the contents in the files
$ cat example2  && echo '' && cat somedir/example
this text will be in the file 'example2'
This file was created by a custom procedure! Neat, huh?
```

```
$ git config -f .datalad/config datalad.procedures.example.help "A toy example"
$ datalad save -m "add help description"
add(ok): .datalad/config (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

To find out more about a given procedure, you can ask for help:

```
$ datalad run-procedure --help-proc example
example (.datalad/procedures/example.py)
A toy example
```

---

[111] Note that we simplify the level of procedures that exist within a dataset by calling them *local*. Even though they apply to a dataset just as *local* Git configurations, unlike Git's *local* configurations in `.git/config`, the procedures and procedure configurations in `.datalad/config` are committed and can be shared together with a dataset. The procedure level *local* therefore does not exactly corresponds to the *local* scope in the sense that Git uses it.

MAKE THE MOST OUT OF DATASETS

11.1 A Data Analysis Project with DataLad

Time flies and the semester rapidly approaches the midterms. In DataLad-101, students are not given an exam – instead, they are asked to complete and submit a data analysis project with DataLad.

The lecturer hands out the requirements: The project...

- needs to be a data analysis project

- is to be prepared in the form of a DataLad dataset

- should incorporate DataLad whenever possible (data retrieval, publication, script execution, general version control) and

- needs to comply to the YODA principles

Luckily, the midterms are only in a couple of weeks, and a lot of the requirements of the project will be taught in the upcoming sessions. Therefore, there's little you can do to prepare for the midterm than to be extra attentive on the next lectures on the YODA principles and DataLad's Python API.

11.2 YODA: Best practices for data analyses in a dataset

The last requirement for the midterm projects reads "needs to comply to the YODA principles". "What are the YODA principles?" you ask, as you have never heard of this before. "The topic of today's lecture: Organizational principles of data analyses in DataLad datasets. This lecture will show you the basic principles behind creating, sharing, and publishing reproducible, understandable, and open data analysis projects with DataLad.", you hear in return.

The starting point…

Data analyses projects are very common, both in science and industry. But it can be very difficult to produce a reproducible, let alone *comprehensible* data analysis project. Many data analysis projects do not start out with a stringent organization, or fail to keep the structural organization of a directory intact as the project develops. Often, this can be due to a lack of version-control. In these cases, a project will quickly end up with many almost-identical scripts suffixed with "_version_xyz"[112], or a chaotic results structure split between various directories with names such as `results/`, `results_August19/`, `results_revision/` and `now_with_nicer_plots/`. Something like this is a very common shape a data science project may take after a while:

```
├── code/
│   ├── code_final/
│   │   ├── final_2/
│   │   │   ├── main_script_fixed.py
│   │   │   └──takethisscriptformostthingsnow.py
│   │   ├── utils_new.py
│   │   ├── main_script.py
│   │   ├── utils_new.py
│   │   ├── utils_2.py
│   │   └── main_analysis_newparameters.py
│   └── main_script_DONTUSE.py
├── data/
│   ├── data_updated/
│   │   └── dataset1/
│   │       ├── datafile_a
│   ├── dataset1/
│   │   ├── datafile_a
│   ├── outputs/
│   │   ├── figures/
│   │   │   ├── figures_new.py
│   │   │   └── figures_final_forreal.py
│   │   ├── important_results/
│   │   ├── random_results_file.tsv
│   │   ├── results_for_paper/
│   │   ├── results_for_paper_revised/
│   │   └── results_new_data/
│   ├── random_results_file.tsv
│   ├── random_results_file_v2.tsv
```

[...]

All data analysis endeavors in directories like this *can* work, for a while, if there is a person who knows the project well, and works on it all the time. But it inevitably will get messy

---

[112] http://phdcomics.com/comics/archive.php?comicid=1531

once anyone tries to collaborate on a project like this, or simply goes on a two-week vacation and forgets whether the function in `main_analysis_newparameters.py` or the one in `takethisscriptformostthingsnow.py` was the one that created a particular figure.

But even if a project has an intuitive structure, and *is* version controlled, in many cases an analysis script will stop working, or maybe worse, will produce different results, because the software and tools used to conduct the analysis in the first place got an update. This update may have come with software changes that made functions stop working, or work differently than before. In the same vein, recomputing an analysis project on a different machine than the one the analysis was developed on can fail if the necessary software in the required versions is not installed or available on this new machine. The analysis might depend on software that runs on a Linux machine, but the project was shared with a Windows user. The environment during analysis development used Python 2, but the new system has only Python 3 installed. Or one of the dependent libraries needs to be in version X, but is installed as version Y.

The YODA principles are a clear set of organizational standards for datasets used for data analysis projects that aim to overcome issues like the ones outlined above. The name stands for "YODAs Organigram on Data Analysis"[120]. The principles outlined in YODA set simple rules for directory names and structures, best-practices for version-controlling dataset elements and analyses, facilitate usage of tools to improve the reproducibility and accountability of data analysis projects, and make collaboration easier. They are summarized in three basic principles, that translate to both dataset structures and best practices regarding the analysis:

- *P1: One thing, one dataset* (page 137)

- *P2: Record where you got it from, and where it is now* (page 140)

- *P3: Record what you did to it, and with what* (page 141)

As you will see, complying to these principles is easy if you use DataLad. Let's go through them one by one:

P1: One thing, one dataset

Whenever a particular collection of files could be useful in more than one context, make them a standalone, modular component. In the broadest sense, this means to structure your study elements (data, code, computational environments, results, . . .) in dedicated directories. For example:

- Store **input data** for an analysis in a dedicated `inputs/` directory. Keep different formats or processing-stages of your input data as individual, modular components: Do not mix raw data, data that is already structured following community guidelines of the given field, or preprocessed data, but create one data component for each of them. And if your analysis relies on two or more data collections, these collections should each be an individual component, not combined into one.

- Store scripts or **code** used for the analysis of data in a dedicated `code/` directory, outside of the data component of the dataset.

---

[120] "Why does the acronym contain itself?" you ask confused. "That's because it's a recursive acronym[121], where the first letter stands recursively for the whole acronym." you get in response. "This is a reference to the recursiveness within a DataLad dataset – all principles apply recursively to all the subdatasets a dataset has." "And what does all of this have to do with Yoda?" you ask mildly amused. "Oh, well. That's just because the DataLad team is full of geeks."

[121] https://en.wikipedia.org/wiki/Recursive_acronym

- Collect **results** of an analysis in a dedicated `outputs/` directory, and leave the input data of an analysis untouched by your computations.

- Include a place for complete **execution environments**, for example singularity images[113] or docker containers[114,122], in the form of an `envs/` directory, if relevant for your analysis.

- And if you conduct multiple different analyses, create a dedicated project for each analysis, instead of conflating them.

This, for example, would be a directory structure from the root of a superdataset of a very comprehensive data analysis project complying to the YODA principles:

```
├── ci/                        # continuous integration configuration
│   └── .travis.yml
├── code/                      # your code
│   ├── tests/                 # unit tests to test your code
│   │   └── test_myscript.py
│   └── myscript.py
├── docs                       # documentation about the project
│   ├── build/
│   └── source/
├── envs                       # computational environments
│   └── Singularity
├── inputs/                    # dedicated inputs/, will not be changed by an analysis
│   └── data/
│       ├── dataset1/          # one stand-alone data component
│       │   └── datafile_a
│       └── dataset2/
│           └── datafile_a
├── outputs/                   # outputs away from the input data
│   └── important_results/
│       └── figures/
├── CHANGELOG.md               # notes for fellow humans about your project
├── HOWTO.md
└── README.md
```

You can get a few non-DataLad related advice for structuring your directories in the *on best practices for analysis organization* (page 139).

There are many advantages to this modular way of organizing contents. Having input data as independent components that are not altered (only consumed) by an analysis does not conflate the data for an analysis with the results or the code, thus assisting understanding the project for anyone unfamiliar with it. But more than just structure, this organization aids modular reuse or publication of the individual components, for example data. In a YODA-compliant dataset, any processing stage of a data component can be reused in a new project or published and shared. The same is true for a whole analysis dataset. At one point you might also write a scientific paper about your analysis in a paper project, and the whole analysis project can easily become a modular component in a paper project, to make sharing paper, code, data, and results easy. The usecase *Writing a reproducible paper* (page 399) contains a step-by-step instruction on how

---

[113] https://singularity.lbl.gov/

[114] https://www.docker.com/get-started

[122] If you want to learn more about Docker and Singularity, or general information about containerized computational environments for reproducible data science, check out this section[123] in the wonderful book The Turing Way[124], a comprehensive guide to reproducible data science, or read about it in section *Computational reproducibility with software containers* (page 166).

[123] https://the-turing-way.netlify.app/reproducible-research/renv/renv-containers.html

[124] https://the-turing-way.netlify.app/welcome

> 💡 **M11.1 More best practices for organizing contents in directories**
>
> The exemplary YODA directory structure is very comprehensive, and displays many best-practices for reproducible data science. For example,
>
> 1. Within code/, it is best practice to add **tests** for the code. These tests can be run to check whether the code still works.
> 2. It is even better to further use automated computing, for example continuous integration (CI) systems[115], to test the functionality of your functions and scripts automatically. If relevant, the setup for continuous integration frameworks (such as Travis[116]) lives outside of code/, in a dedicated ci/ directory.
> 3. Include **documents for fellow humans**: Notes in a README.md or a HOWTO.md, or even proper documentation (for example using in a dedicated docs/ directory. Within these documents, include all relevant metadata for your analysis. If you are conducting a scientific study, this might be authorship, funding, change log, etc.
>
> If writing tests for analysis scripts or using continuous integration is a new idea for you, but you want to learn more, check out this chapter on testing[117].
>
> ---
> [115] https://en.wikipedia.org/wiki/Continuous_integration
> [116] https://travis-ci.org
> [117] https://the-turing-way.netlify.app/reproducible-research/testing

to build and share such a reproducible paper, if you want to learn more.
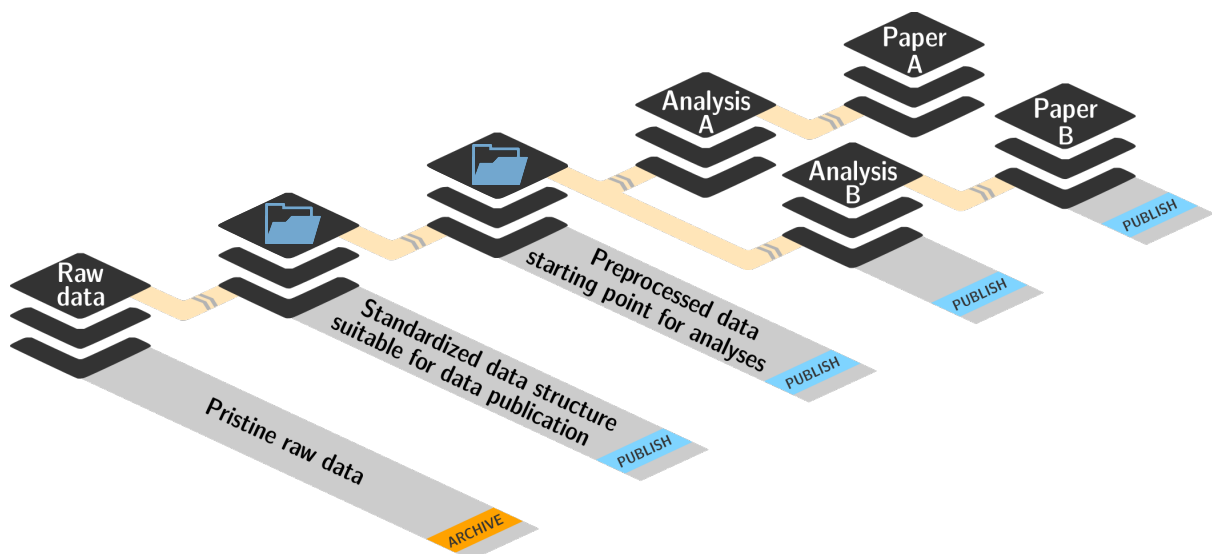
Fig. 11.1: Data are modular components that can be re-used easily.

The directory tree above and Figure 3 highlight different aspects of this principle. The directory tree illustrates the structure of the individual pieces on the file system from the point of view of a single top-level dataset with a particular purpose. It for example could be an analysis dataset created by a statistician for a scientific project, and it could be shared between collaborators or with others during development of the project. In this superdataset, code is created that operates on input data to compute outputs, and the code and outputs are captured, version-controlled, and linked to the input data. Each input data in turn is a (potentially nested) subdataset, but this is not visible in the directory hierarchy. Figure 3 in comparison emphasizes a process view on a project and the nested structure of input subdataset: You can see how the preprocessed data that serves as an input for the analysis datasets evolves from raw data to standardized data organization to its preprocessed state. Within the data/ directory of

---

the file system hierarchy displayed above one would find data datasets with their previous version as a subdataset, and this is repeated recursively until one reaches the raw data as it was originally collected at one point. A finished analysis project in turn can be used as a component (subdataset) in a paper project, such that the paper is a fully reproducible research object that shares code, analysis results, and data, as well as the history of all of these components.

Principle 1, therefore, encourages to structure data analysis projects in a clear and modular fashion that makes use of nested DataLad datasets, yielding comprehensible structures and reusable components. Having each component version-controlled – regardless of size – will aid keeping directories clean and organized, instead of piling up different versions of code, data, or results.

P2: Record where you got it from, and where it is now

It is good to have data, but it is even better if you and anyone you collaborate or share the project or its components with can find out where the data came from, or how it is dependent on or linked to other data. Therefore, this principle aims to attach this information, the data's PROVENANCE, to the components of your data analysis project.

Luckily, this is a no-brainer with DataLad, because the core data structure of DataLad, the dataset, and many of the DataLad commands already covered up to now fulfill this principle.

If data components of a project are DataLad datasets, they can be included in an analysis superdataset as subdatasets. Thanks to **datalad clone**, information on the source of these subdatasets is stored in the history of the analysis superdataset, and they can even be updated from those sources if the original data dataset gets extended or changed. If you are including a file, for example code from GitHub, the **datalad download-url** command (introduced in section *Populate a dataset* (page 37)) will record the source of it safely in the dataset's history. And if you add anything to your dataset, from simple incremental coding progress in your analysis scripts up to files that a colleague sent you via email, a plain **datalad save** with a helpful commit message goes a very long way to fulfill this principle on its own already.

One core aspect of this principle is *linking* between re-usable data resource units (i.e., DataLad subdatasets containing pure data). You will be happy to hear that this is achieved by simply installing datasets as subdatasets. This part of this principle will therefore be absolutely obvious to you because you already know how to install and nest datasets within datasets. "I might just overcome my impostor syndrome if I experience such advanced reproducible analysis concepts as being obvious", you think with a grin.
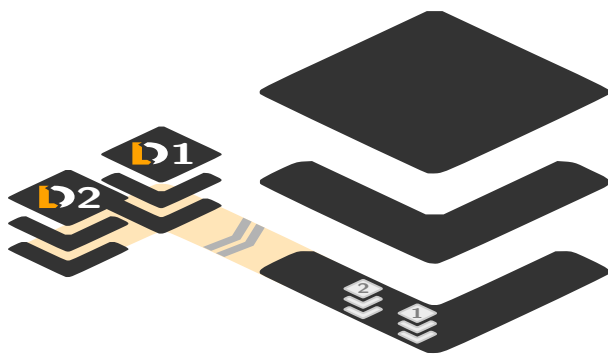

Fig. 11.2: Schematic illustration of two standalone data datasets installed as subdatasets into an analysis project.

But more than linking datasets in a superdataset, linkage also needs to be established between

components of your dataset. Scripts inside of your code/ directory should point to data not as ABSOLUTE PATHs that would only work on your system, but instead as RELATIVE PATHs that will work in any shared copy of your dataset. The next section demonstrates a YODA data analysis project and will show concrete examples of this.

Lastly, this principle also includes *moving*, *sharing*, and *publishing* your datasets or its components. It is usually costly to collect data, and economically unfeasible[125] to keep it locked in a drawer (or similarly out of reach behind complexities of data retrieval or difficulties in understanding the data structure). But conducting several projects on the same dataset yourself, sharing it with collaborators, or publishing it is easy if the project is a DataLad dataset that can be installed and retrieved on demand, and is kept clean from everything that is not part of the data according to principle 1. Conducting transparent open science is easier if you can link code, data, and results within a dataset, and share everything together. In conjunction with principle 1, this means that you can distribute your analysis projects (or parts of it) in a comprehensible form.
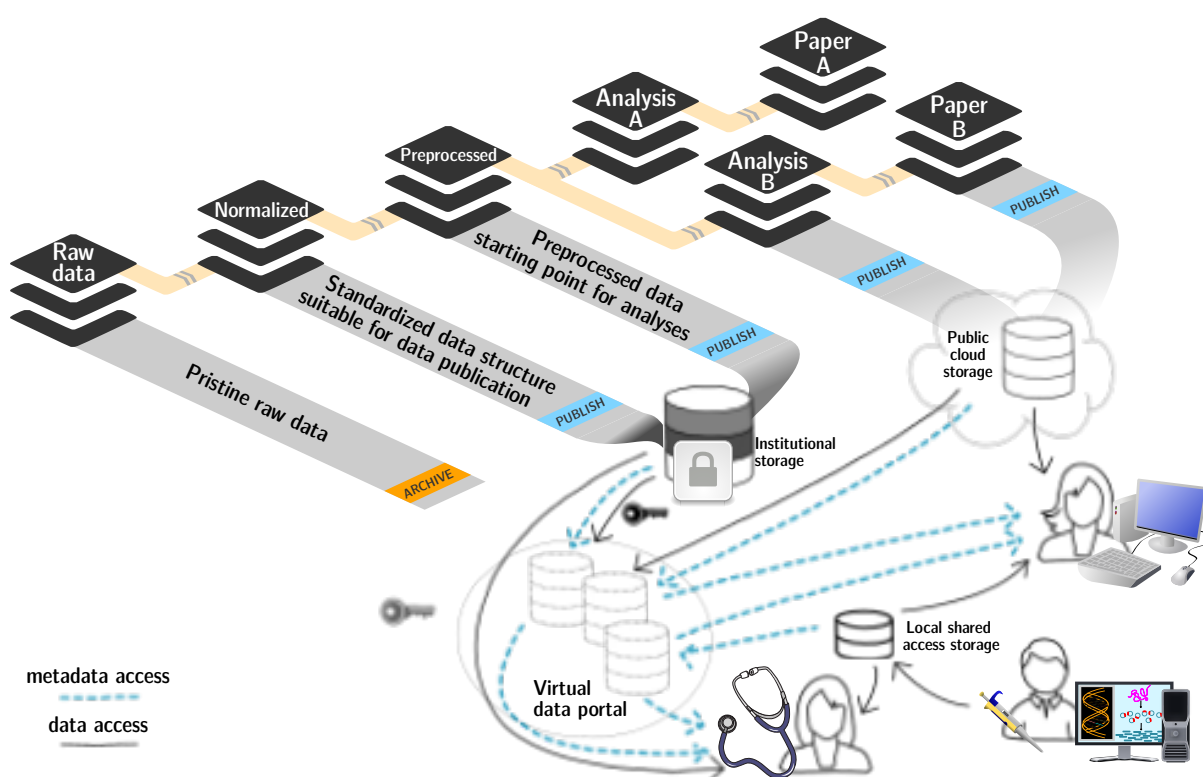
Fig. 11.3: In a dataset that complies to the YODA principles, modular components (data, analysis results, papers) can be shared or published easily.

Principle 2, therefore, facilitates transparent linkage of datasets and their components to other components, their original sources, or shared copies. With the DataLad tools you learned to master up to this point, you have all the necessary skills to comply to it already.

P3: Record what you did to it, and with what

This last principle is about capturing *how exactly the content of every file came to be* that was not obtained from elsewhere. For example, this relates to results generated from inputs by scripts or commands. The section *Keeping track* (page 59) already outlined the problem of associating

---

[125] Substitute unfeasible with *wasteful*, *impractical*, or simply *stupid* if preferred.

a result with an input and a script. It can be difficult to link a figure from your data analysis project with an input data file or a script, even if you created this figure yourself. The **datalad run** command however mitigates these difficulties, and captures the provenance of any output generated with a datalad run call in the history of the dataset. Thus, by using **datalad run** in analysis projects, your dataset knows which result was generated when, by which author, from which inputs, and by means of which command.

With another DataLad command one can even go one step further: The command **datalad containers-run** (it will be introduced in section *Computational reproducibility with software containers* (page 166)) performs a command execution within a configured containerized environment. Thus, not only inputs, outputs, command, time, and author, but also the *software environment* are captured as provenance of a dataset component such as a results file, and, importantly, can be shared together with the dataset in the form of a software container.

Tip: Make use of datalad run's --dry-run option to craft your run-command (see *Dry-running your run call* (page 78))!

With this last principle, your dataset collects and stores provenance of all the contents you created in the wake of your analysis project. This established trust in your results, and enables others to understand where files derive from.

The YODA procedure

There is one tool that can make starting a yoda-compliant data analysis easier: DataLad's yoda procedure. Just as the text2git procedure from section *Create a dataset* (page 34), the yoda procedure can be included in a **datalad create** command and will apply useful configurations to your dataset:

```
$ datalad create -c yoda "my_analysis"

[INFO   ] Creating a new annex repo at /home/me/repos/testing/my_analysis
create(ok): /home/me/repos/testing/my_analysis (dataset)
[INFO   ] Running procedure cfg_yoda
[INFO   ] == Command start (output follows) =====
[INFO   ] == Command exit (modification check follows) =====
```

Let's take a look at what configurations and changes come with this procedure:

```
$ tree -a


.
├── .gitattributes
├── CHANGELOG.md
├── code
│   ├── .gitattributes
│   └── README.md
└── README.md
```

Let's take a closer look into the .gitattributes files:

```
$ less .gitattributes

**/.git* annex.largefiles=nothing
CHANGELOG.md annex.largefiles=nothing
README.md annex.largefiles=nothing
```

(continues on next page)

```
$ less code/.gitattributes

* annex.largefiles=nothing
```

Summarizing these two glimpses into the dataset, this configuration has

1. included a code directory in your dataset

2. included three files for human consumption (`README.md`, `CHANGELOG.md`)

3. configured everything in the `code/` directory to be tracked by Git, not git-annex[126]

4. and configured `README.md` and `CHANGELOG.md` in the root of the dataset to be tracked by Git.

Your next data analysis project can thus get a head start with useful configurations and the start of a comprehensible directory structure by applying the yoda procedure.

Sources

This section is based on this comprehensive poster[118] and these publicly available slides[119] about the YODA principles.

11.3 YODA-compliant data analysis projects

Now that you know about the YODA principles, it is time to start working on `DataLad-101`'s midterm project. Because the midterm project guidelines require a YODA-compliant data analysis project, you will not only have theoretical knowledge about the YODA principles, but also gain practical experience.

In principle, you can prepare YODA-compliant data analyses in any programming language of your choice. But because you are already familiar with the Python[127] programming language, you decide to script your analysis in Python. Delighted, you find out that there is even a Python API for DataLad's functionality that you can read about in *a Findoutmore on DataLad in Python* (page 158).

> ⚠️ **Use DataLad in languages other than Python**
>
> While there is a dedicated API for Python, DataLad's functions can of course also be used with other programming languages, such as Matlab, via standard system calls.
> Even if you do not know or like Python, you can just copy-paste the code and follow along – the high-level YODA principles demonstrated in this section generalize across programming languages.

---

[126] To re-read how `.gitattributes` work, go back to section *DIY configurations* (page 112), and to remind yourself about how this worked for the `text2git` configuration, go back to section *Data safety* (page 83).

[118] https://f1000research.com/posters/7-1965

[119] https://github.com/myyoda/talk-principles

[127] https://www.python.org/

For your midterm project submission, you decide to create a data analysis on the iris flower data set[131]. It is a multivariate dataset on 50 samples of each of three species of Iris flowers (*Setosa*, *Versicolor*, or *Virginica*), with four variables: the length and width of the sepals and petals of the flowers in centimeters. It is often used in introductory data science courses for statistical classification techniques in machine learning, and widely available – a perfect dataset for your midterm project!

⚠️ **Turn data analysis into dynamically generated documents**

Beyond the contents of this section, we have transformed the example analysis also into a template to write a reproducible paper, following the use case *Writing a reproducible paper* (page 399). If you're interested in checking that out, please head over to github.com/datalad-handbook/repro-paper-sketch/[132].

---

[132] https://github.com/datalad-handbook/repro-paper-sketch/

Raw data as a modular, independent entity

The first YODA principle stressed the importance of modularity in a data analysis project: Every component that could be used in more than one context should be an independent component.

The first aspect this applies to is the input data of your dataset: There can be thousands of ways to analyze it, and it is therefore immensely helpful to have a pristine raw iris dataset that does not get modified, but serves as input for these analysis. As such, the iris data should become a standalone DataLad dataset. For the purpose of this analysis, the DataLad handbook provides an `iris_data` dataset at https://github.com/datalad-handbook/iris_data.

You can either use this provided input dataset, or find out how to create an independent dataset from scratch in a *dedicated Findoutmore* (page 160).

"Nice, with this input dataset I have sufficient provenance capture for my input dataset, and I can install it as a modular component", you think as you mentally tick off YODA principle number 1 and 2. "But before I can install it, I need an analysis superdataset first."

Building an analysis dataset

There is an independent raw dataset as input data, but there is no place for your analysis to live, yet. Therefore, you start your midterm project by creating an analysis dataset. As this project is part of `DataLad-101`, you do it as a subdataset of `DataLad-101`. Remember to specify the `--dataset` option of **datalad create** to link it as a subdataset!

You naturally want your dataset to follow the YODA principles, and, as a start, you use the `cfg_yoda` procedure to help you structure the dataset[145]:

---

[131] https://en.wikipedia.org/wiki/Iris_flower_data_set

[145] Note that you could have applied the YODA procedure not only right at creation of the dataset with `-c yoda`, but also after creation with the **datalad run-procedure** command:

```
$ cd midterm_project
$ datalad run-procedure cfg_yoda
```

Both ways of applying the YODA procedure will lead to the same outcome.

```
# inside of DataLad-101
$ datalad create -c yoda --dataset . midterm_project
[INFO] Creating a new annex repo at /home/me/dl-101/DataLad-101/midterm_project
[INFO] scanning for unlocked files (this may take some time)
[INFO] Running procedure cfg_yoda
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): midterm_project (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): midterm_project (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)
```

The **datalad subdatasets** command can report on which subdatasets exist for DataLad-101.
This helps you verify that the command succeeded and the dataset was indeed linked as a
subdataset to DataLad-101:

```
$ datalad subdatasets
subdataset(ok): midterm_project (dataset)
subdataset(ok): recordings/longnow (dataset)
action summary:
  subdataset (ok: 2)
```

Not only the longnow subdataset, but also the newly created midterm_project subdataset are
displayed – wonderful!

But back to the midterm project now. So far, you have created a pre-structured analysis dataset.
As a next step, you take care of installing and linking the raw dataset for your analysis ade-
quately to your midterm_project dataset by installing it as a subdataset. Make sure to install it
as a subdataset of midterm_project, and not DataLad-101!

```
$ cd midterm_project
# we are in midterm_project, thus -d . points to the root of it.
$ datalad clone -d . \
  https://github.com/datalad-handbook/iris_data.git \
  input/
[INFO] Cloning dataset to Dataset(/home/me/dl-101/DataLad-101/midterm_project/input)
[INFO] Attempting to clone from https://github.com/datalad-handbook/iris_data.git to /
→home/me/dl-101/DataLad-101/midterm_project/input
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/DataLad-101/midterm_project/
→input)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Remote origin not usable by git-annex; setting annex-ignore
install(ok): input (dataset)
add(ok): input (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
```

<div align="right">(continues on next page)</div>

```
add (ok: 2)
install (ok: 1)
save (ok: 1)
```

Note that we did not keep its original name, `iris_data`, but rather provided a path with a new name, `input`, because this much more intuitively comprehensible.

After the input dataset is installed, the directory structure of `DataLad-101` looks like this:

```
$ cd ../
$ tree -d
$ cd midterm_project
.
├── books
├── code
├── midterm_project
│   ├── code
│   └── input
└── recordings
    └── longnow
        ├── Long_Now__Conversations_at_The_Interval
        └── Long_Now__Seminars_About_Long_term_Thinking

9 directories
```

Importantly, all of the subdatasets are linked to the higher-level datasets, and despite being inside of `DataLad-101`, your `midterm_project` is an independent dataset, as is its `input/` subdataset:
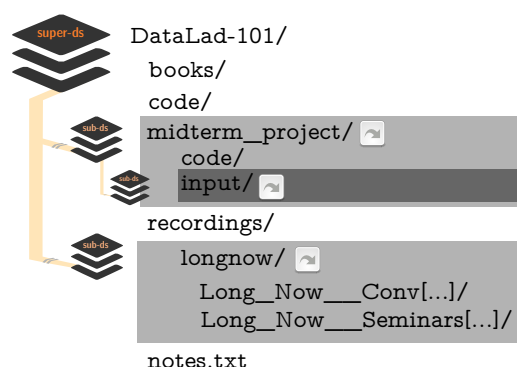
YODA-compliant analysis scripts

Now that you have an `input/` directory with data, and a `code/` directory (created by the YODA procedure) for your scripts, it is time to work on the script for your analysis. Within `midterm_project`, the `code/` directory is where you want to place your scripts. Finally you can try out the Python API of DataLad!

But first, you plan your research question. You decide to do a classification analysis with a k-nearest neighbors algorithm[146]. The iris dataset works well for such questions. Based on the

---

[146] If you want to know more about this algorithm, this blogpost[147] gives an accessible overview. However, the choice of analysis method for the handbook is rather arbitrary, and understanding the k-nearest neighbor algorithm is by no means required for this section.

[147] https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d017

features of the flowers (sepal and petal width and length) you will try to predict what type of flower (*Setosa*, *Versicolor*, or *Virginica*) a particular flower in the dataset is. You settle on two objectives for your analysis:

1. Explore and plot the relationship between variables in the dataset and save the resulting graphic as a first result.

2. Perform a k-nearest neighbor classification on a subset of the dataset to predict class membership (flower type) of samples in a left-out test set. Your final result should be a statistical summary of this prediction.

To compute the analysis you create the following Python script inside of code/:

```
$ cat << EOT > code/script.py

import pandas as pd
import seaborn as sns
import datalad.api as dl
from sklearn import model_selection
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report

data = "input/iris.csv"

# make sure that the data are obtained (get will also install linked sub-ds!):
dl.get(data)

# prepare the data as a pandas dataframe
df = pd.read_csv(data)
attributes = ["sepal_length", "sepal_width", "petal_length","petal_width", "class"]
df.columns = attributes

# create a pairplot to plot pairwise relationships in the dataset
plot = sns.pairplot(df, hue='class', palette='muted')
plot.savefig('pairwise_relationships.png')

# perform a K-nearest-neighbours classification with scikit-learn
# Step 1: split data in test and training dataset (20:80)
array = df.values
X = array[:,0:4]
Y = array[:,4]
test_size = 0.20
seed = 7
X_train, X_test, Y_train, Y_test = model_selection.train_test_split(X, Y,
                                                        test_size=test_size,
                                                        random_state=seed)

# Step 2: Fit the model and make predictions on the test dataset
knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)
predictions = knn.predict(X_test)

# Step 3: Save the classification report
report = classification_report(Y_test, predictions, output_dict=True)
df_report = pd.DataFrame(report).transpose().to_csv('prediction_report.csv')

EOT
```

61

This script will

- import DataLad's functionality and expose it as dl.<COMMAND>

- make sure to install the linked subdataset and retrieve the data with **datalad get** (l. 12) prior to reading it in, and

- save the resulting figure (l. 21) and .csv file (l. 40) into the root of midterm_project/. Note how this helps to fulfil YODA principle 1 on modularity: Results are stored outside of the pristine input subdataset.

- Note further how all paths (to input data and output files) are *relative*, such that the midterm_project analysis is completely self-contained within the dataset, contributing to fulfill the second YODA principle.

Let's run a quick **datalad status**. . .

```
$ datalad status
untracked: code/script.py (file)
```

. . . and save the script to the subdataset's history. As the script completes your analysis setup, we *tag* the state of the dataset to refer to it easily at a later point with the --version-tag option of **datalad save**.

```
$ datalad save -m "add script for kNN classification and plotting" \
  --version-tag ready4analysis \
  code/script.py
add(ok): code/script.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

> 💡 **M11.4 What is a tag?**
>
> TAGs are markers that you can attach to commits in your dataset history. They can have any name, and can help you and others to identify certain commits or dataset states in the history of a dataset. Let's take a look at how the tag you just created looks like in your history with **git show**. Note how we can use a tag just as easily as a commit SHASUM:
>
> ```
> $ git show ready4analysis
> commit dd364728c767a7ef3be5b1df229901a27f1cec6d
> Author: Elena Piscopia <elena@example.net>
> Date:   Thu Jul 29 16:22:38 2021 +0200
>
>     add script for kNN classification and plotting
>
> diff --git a/code/script.py b/code/script.py
> new file mode 100644
> index 0000000..26058d3
> --- /dev/null
> +++ b/code/script.py
> @@ -0,0 +1,41 @@
> ```
>
> This tag thus identifies the version state of the dataset in which this script was added. Later we can use this tag to identify the point in time at which the analysis setup was ready – much more intuitive than a 40-character shasum! This is handy in the context of

> 💡 a **datalad rerun** for example:
>
> ```
> $ datalad rerun --since ready4analysis
> ```
>
> would rerun any **run** command in the history performed between tagging and the current dataset state.

Finally, with your directory structure being modular and intuitive, the input data installed, the script ready, and the dataset status clean, you can wrap the execution of the script (which is a simple `python3 code/script.py`) in a **datalad run** command. Note that simply executing the script would work as well – thanks to DataLad's Python API. But using **datalad run** will capture full provenance, and will make re-execution with **datalad rerun** easy.

> ⚠️ **Additional software requirements: pandas, seaborn, sklearn**
>
> Note that you need to have the following Python packages installed to run the analysis[148]:
> - pandas[135]
> - seaborn[136]
> - sklearn[137]
>
> The packages can be installed via `pip`. Check the footnote[148] for code snippets to copy and paste. However, if you do not want to install any Python packages, do not execute the remaining code examples in this section – an upcoming section on `datalad containers-run` will allow you to perform the analysis without changing your Python software-setup.
>
> ---
>
> [148] It is recommended (but optional) to create a virtual environment[149] and install the required Python packages inside of it:
>
> ```
> # create and enter a new virtual environment (optional)
> $ virtualenv --python=python3 ~/env/handbook
> $ . ~/env/handbook/bin/activate
> ```
>
> ```
> # install the Python packages from PyPi via pip
> pip install seaborn pandas sklearn
> ```
>
> [135] https://pandas.pydata.org/
> [136] https://seaborn.pydata.org/
> [137] https://scikit-learn.org/

> 🪟 **W11.1 You may need to use "python", not "python3"**
>
> If executing the code below returns an exit code of 9009, there may be no `python3` – instead, it is called solely `python`. Please run the following instead (adjusted for line breaks, you should be able to copy-paste this as a whole):
>
> ```
> datalad run -m "analyze iris data with classification analysis" ^
>  --input "input/iris.csv" ^
>  --output "prediction_report.csv" ^
>  --output "pairwise_relationships.png" ^
>  "python code/script.py"
> ```

```
$ datalad run -m "analyze iris data with classification analysis" \
  --input "input/iris.csv" \
  --output "prediction_report.csv" \
```

(continues on next page)

```
  --output "pairwise_relationships.png" \
  "python3 code/script.py"
[INFO] Making sure inputs are available (this may take some time)
get(ok): input/iris.csv (file) [from web...]
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): pairwise_relationships.png (file)
add(ok): prediction_report.csv (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 2, ok: 1)
  save (notneeded: 1, ok: 1)
```

As the successful command summary indicates, your analysis seems to work! Two files were created and saved to the dataset: `pairwise_relationships.png` and `prediction_report.csv`. If you want, take a look and interpret your analysis. But what excites you even more than a successful data science project on first try is that you achieved complete provenance capture:

- Every single file in this dataset is associated with an author and a time stamp for each modification thanks to **datalad save**.

- The raw dataset knows where the data came from thanks to **datalad clone** and **datalad download-url**.

- The subdataset is linked to the superdataset thanks to **datalad clone -d**.

- The **datalad run** command took care of linking the outputs of your analysis with the script and the input data it was generated from, fulfilling the third YODA principle.

Let's take a look at the history of the `midterm_project` analysis dataset:

```
$ git log --oneline
f76acca [DATALAD RUNCMD] analyze iris data with classification analysis
dd36472 add script for kNN classification and plotting
01447d9 [DATALAD] modified subdataset properties
2465103 [DATALAD] Recorded changes
261ea45 Apply YODA dataset setup
50e40b2 [DATALAD] new dataset
```

"Wow, this is so clean an intuitive!" you congratulate yourself. "And I think this was and will be the fastest I have ever completed a midterm project!" But what is still missing is a human readable description of your dataset. The YODA procedure kindly placed a `README.md` file into the root of your dataset that you can use for this[150].

> ⚠️ **Template for introduction to DataLad**
>
> If you plan to share your own datasets with people that are unfamiliar with DataLad, it may be helpful to give a short explanation of what a DataLad dataset is and what it can do. For this, you can use a ready-made text block that the handbook provides. To find this textblock, go to *How can I help others get started with a shared dataset?* (page 482).

---

[150] Note that all `README.md` files the YODA procedure created are version controlled by Git, not git-annex, thanks to the configurations that YODA supplied. This makes it easy to change the `README.md` file. The previous section detailed how the YODA procedure configured your dataset. If you want to re-read the full chapter on configurations and run-procedures, start with section *DIY configurations* (page 112).

```
# with the >| redirection we are replacing existing contents in the file
$ cat << EOT >| README.md

# Midterm YODA Data Analysis Project

## Dataset structure

- All inputs (i.e. building blocks from other sources) are located in input/.
- All custom code is located in code/.
- All results (i.e., generated files) are located in the root of the dataset:
  - "prediction_report.csv" contains the main classification metrics.
  - "output/pairwise_relationships.png" is a plot of the relations between features.

EOT


$ datalad status
 modified: README.md (file)


$ datalad save -m "Provide project description" README.md
add(ok): README.md (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Note that one feature of the YODA procedure was that it configured certain files (for example everything inside of code/, and the README.md file in the root of the dataset) to be saved in Git instead of git-annex. This was the reason why the README.md in the root of the dataset was easily modifiable[150].

> **M11.5 Saving contents with Git regardless of configuration with –to-git**
>
> The yoda procedure in midterm_project applied a different configuration within .gitattributes than the text2git procedure did in DataLad-101. Within DataLad-101, any text file is automatically stored in GIT. This is not true in midterm_project: Only the existing README.md files and anything within code/ are stored – everything else will be annexed. That means that if you create any other file, even text files, inside of midterm_project (but not in code/), it will be managed by GIT-ANNEX and content-locked after a **datalad save** – an inconvenience if it would be a file that is small enough to be handled by Git.
>
> Luckily, there is a handy shortcut to saving files in Git that does not require you to edit configurations in .gitattributes: The --to-git option for **datalad save**.
>
> ```
> $ datalad save -m "add sometextfile.txt" --to-git sometextfile.txt
> ```

After adding this short description to your README.md, your dataset now also contains sufficient human-readable information to ensure that others can understand everything you did easily. The only thing left to do is to hand in your assignment. According to the syllabus, this should be done via GITHUB.

> 💡 **M11.6 What is GitHub?**
>
> GitHub is a web based hosting service for Git repositories. Among many different other useful perks it adds features that allow collaboration on Git repositories. GitLab[138] is a similar service with highly similar features, but its source code is free and open, whereas GitHub is a subsidiary of Microsoft.
>
> Web-hosting services like GitHub and GITLAB integrate wonderfully with DataLad. They are especially useful for making your dataset publicly available, if you have figured out storage for your large files otherwise (as large content can not be hosted for free by GitHub). You can make DataLad publish large file content to one location and afterwards automatically push an update to GitHub, such that users can install directly from GitHub/GitLab and seemingly also obtain large file content from GitHub. GitHub can also resolve subdataset links to other GitHub repositories, which lets you navigate through nested datasets in the web-interface.
>
>
>
> The above screenshot shows the linkage between the analysis project you will create and its subdataset. Clicking on the subdataset (highlighted) will take you to the iris dataset the handbook provides, shown below.
>
>
>
> ---
>
> [138] https://about.gitlab.com/

Publishing the dataset to GitHub

> ⚠️ **Demo needs a GitHub account or alternative**
>
> The upcoming part requires a GitHub account. If you do not have one you can either
> - Create one now – it is fast, free, and you can get rid of it afterwards, if you want to.
> - Or exchange the command `create-sibling-github` with `create-sibling-gitlab` if you have a GitLab account instead of a GitHub account (checkout the documentation[139] for differences in invocation beforehand, though).
> - Decide to not follow along.
>
> ---
> [139] http://docs.datalad.org/en/stable/generated/man/datalad-create-sibling-gitlab.html

For this, you need to

- create a repository for this dataset on GitHub,

- configure this GitHub repository to be a SIBLING of the `midterm_project` dataset,

- and *publish* your dataset to GitHub.

Luckily, DataLad can make all of this very easy with the **datalad create-sibling-github** (datalad-create-sibling-github manual) command (or, for GitLab[140], **datalad create-sibling-gitlab**, datalad-create-sibling-gitlab manual).

The two commands have different arguments and options. Here, we look at **datalad create-sibling-github**. The command takes a repository name and GitHub authentication credentials (either in the command line call with options `github-login <NAME>` and `github-passwd <PASSWORD>`, with an *oauth* token[141] stored in the Git configuration, or interactively).

> ⚠️ **GitHub deprecated User Password authentication**
>
> GitHub decided to deprecate user-password authentication[142] and will only support authentication via personal access token from November 13th 2020 onwards. Upcoming changes in DataLad's API will reflect this change starting with DataLad version `0.13.6` by removing the `github-passwd` argument. Starting with DataLad `0.16.0`, a new set of commands for interactions with a variety of hosting services will be introduced (for more information, see section *Publishing datasets to Git repository hosting* (page 184)).
> To ensure successful authentication, please create a personal access token at github.com/settings/tokens[143,151], and either
> - supply the token with the argument `--github-login <TOKEN>` from the command line,
> - or supply the token from the command line when queried for a password
>
> ---
> [142] https://developer.github.com/changes/2020-02-14-deprecating-password-auth/
> [143] https://github.com/settings/tokens
> [151] Instead of using GitHub's WebUI you could also obtain a token using the command line GitHub interface (https://github.com/sociomantic-tsunami/git-hub) by running `git hub setup` (if no 2FA is used). If you decide to use the command line interface, here is help on how to use it: Clone the GitHub repository[152] to your local computer. Decide whether you want to build a Debian package to install, or install the single-file Python script distributed in the repository. Make sure that all requirements[153] for your preferred version are installed , and run either `make deb` followed by `sudo dpkg -i deb/git-hub*all.deb`, or `make`

---
[140] https://about.gitlab.com/
[141] https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token

> ⚠️
> ```
>     install.
> ```

Based on the credentials and the repository name, it will create a new, empty repository on GitHub, and configure this repository as a sibling of the dataset:

> **W11.2 Your shell will not display credentials**
>
> Don't be confused if you are prompted for your GitHub credentials, but can't seem to type – the terminal protects your private information by not displaying what you type. Simply type in what is requested, and press enter.

```
$ datalad create-sibling-github -d . midtermproject
.: github(-) [https://github.com/adswa/midtermproject.git (git)]
'https://github.com/adswa/midtermproject.git' configured as sibling 'github' for <Dataset␣
↪path=/home/me/dl-101/DataLad-101/midterm_project>
```

Verify that this worked by listing the siblings of the dataset:

```
$ datalad siblings
[WARNING] Failed to determine if github carries annex.
.: here(+) [git]
.: github(-) [https://github.com/adswa/midtermproject.git (git)]
```

> **G11.1 Create-sibling-github internals**
>
> Creating a sibling on GitHub will create a new empty repository under the account that you provide and set up a *remote* to this repository. Upon a `datalad push` to this sibling, your datasets history will be pushed there.

On GitHub, you will see a new, empty repository with the name midtermproject. However, the repository does not yet contain any of your dataset's history or files. This requires *publishing* the current state of the dataset to this SIBLING with the `datalad push` (datalad-push manual) command.

> ⚠️ **Learn how to push "on the job"**
>
> Publishing is one of the remaining big concepts that this handbook tries to convey. However, publishing is a complex concept that encompasses a large proportion of the previous handbook content as a prerequisite. In order to be not too overwhelmingly detailed, the upcoming sections will approach **push** from a "learning-by-doing" perspective: You will see a first **push** to GitHub below, and the *Findoutmore on the published dataset* (page 161) at the end of this section will already give a practical glimpse into the difference between annexed contents and contents stored in Git when pushed to GitHub. The chapter *Third party infrastructure* (page 177) will extend on this, but the section *Overview: The datalad push command* (page 219) will finally combine and link all the previous contents to give a comprehensive and detailed wrap up of the concept of publishing datasets. In this section, you will also find a detailed overview on how **push** works and which options are available. If you are impatient or need an overview on publishing, feel free to skip ahead. If you have time to follow along, reading the next sections will get you towards a complete picture of publishing a bit more small-stepped and gently. For now, we will start with learning by doing, and the fundamental basics of `datalad push`: The command will make the last saved state of your dataset available (i.e., publish it) to the SIBLING

⚠️ you provide with the `--to` option.

```
$ datalad push --to github
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Transfer data
copy(ok): pairwise_relationships.png (file) [to github...]
copy(ok): prediction_report.csv (file) [to github...]
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
publish(ok): . (dataset) [refs/heads/git-annex->github:refs/heads/git-annex 87cf726..
↪17179ba]
publish(ok): . (dataset) [refs/heads/master->github:refs/heads/master [new branch]]
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101/midterm_project)
```

Thus, you have now published your dataset's history to a public place for others to see and clone. Below we will explore how this may look and feel for others.

⚠️ **Cave! Your default branch may be git-annex**

> If your published dataset looks weird, with cryptic directories names instead of file names, GitHub may have made the GIT-ANNEX BRANCH your repositories' default branch. Learn how to fix this in the corresponding *FAQ* (page 487).

There is one important detail first, though: By default, your tags will not be published. Thus, the tag `ready4analysis` is not pushed to GitHub, and currently this version identifier is unavailable to anyone else but you. The reason for this is that tags are viral – they can be removed locally, and old published tags can cause confusion or unwanted changes. In order to publish a tag, an additional **git push** with the `--tags` option is required:

```
$ git push github --tags
```

G11.2 Pushing tags

> Note that this is a **git push**, not **datalad push**. Tags could be pushed upon a **datalad push**, though, if one configures (what kind of) tags to be pushed. This would need to be done on a per-sibling basis in `.git/config` in the `remote.*.push` configuration. If you had a SIBLING "github", the following configuration would push all tags that start with a v upon a **datalad push --to github**:
>
> ```
> $ git config --local remote.github.push 'refs/tags/v*'
> ```
>
> This configuration would result in the following entry in `.git/config`:
>
> ```
> [remote "github"]
>         url = git@github.com/adswa/midtermproject.git
>         fetch = +refs/heads/*:refs/remotes/github/*
>         annex-ignore = true
>         push = refs/tags/v*
> ```

Yay! Consider your midterm project submitted! Others can now install your dataset and check

---

out your data science project – and even better: they can reproduce your data science project easily from scratch (take a look into the *Findoutmore* (page 161) to see how)!

> **G11.3 Push internals**
>
> The **datalad push** uses `git push`, and `git annex copy` under the hood. Publication targets need to either be configured remote Git repositories, or git-annex special remotes (if they support data upload).

11.4 Summary

The YODA principles are a small set of guidelines that can make a huge difference towards reproducibility, comprehensibility, and transparency in a data analysis project. By applying them in your own midterm analysis project, you have experienced their immediate benefits.

You also noticed that these standards are not complex – quite the opposite, they are very intuitive. They structure essential components of a data analysis project – data, code, potentially computational environments, and lastly also the results – in a modular and practical way, and use basic principles and commands of DataLad you are already familiar with.

There are many advantages to this organization of contents.

- Having input data as independent dataset(s) that are not influenced (only consumed) by an analysis allows for a modular reuse of pure data datasets, and does not conflate the data of an analysis with the results or the code. You have experienced this with the `iris_data` subdataset.

- Keeping code within an independent, version-controlled directory, but as a part of the analysis dataset, makes sharing code easy and transparent, and helps to keep directories neat and organized. Moreover, with the data as subdatasets, data and code can be automatically shared together. By complying to this principle, you were able to submit both code and data in a single superdataset.

- Keeping an analysis dataset fully self-contained with relative instead of absolute paths in scripts is critical to ensure that an analysis reproduces easily on a different computer.

- DataLad's Python API makes all of DataLad's functionality available in Python, either as standalone functions that are exposed via `datalad.api`, or as methods of the `Dataset` class. This provides an alternative to the command line, but it also opens up the possibility of performing DataLad commands directly inside of scripts.

- Including the computational environment into an analysis dataset encapsulates software and software versions, and thus prevents re-computation failures (or sudden differences in the results) once software is updated, and software conflicts arising on different machines than the one the analysis was originally conducted on. You have not yet experienced how to do this first-hand, but you will in a later section.

- Having all of these components as part of a DataLad dataset allows version controlling all pieces within the analysis regardless of their size, and generates provenance for everything, especially if you make use of the tools that DataLad provides. This way, anyone can understand and even reproduce your analysis without much knowledge about your project.

- The yoda procedure is a good starting point to build your next data analysis project up on.

Now what can I do with it?

Using tools that DataLad provides you are able to make the most out of your data analysis project. The YODA principles are a guide to accompany you on your path to reproducibility and provenance-tracking.

What should have become clear in this section is that you are already equipped with enough DataLad tools and knowledge that complying to these standards felt completely natural and effortless in your midterm analysis project.

> 💡 **M11.2 DataLad's Python API**
>
> "Whatever you can do with DataLad from the command line, you can also do it with DataLad's Python API", begins the lecturer. "In addition to the command line interface you are already very familiar with, DataLad's functionality can also be used within interactive Python sessions or Python scripts. This feature can help to automate dataset operations, provides an alternative to the command line, and it is immensely useful when creating reproducible data analyses."
>
> All of DataLad's user-oriented commands are exposed via `datalad.api`. Thus, any command can be imported as a stand-alone command like this:
>
> ```
> >>> from datalad.api import <COMMAND>
> ```
>
> Alternatively, to import all commands, one can use
>
> ```
> >>> import datalad.api as dl
> ```
>
> and subsequently access commands as `dl.get()`, `dl.clone()`, and so forth. The developer documentation[128] of DataLad lists an overview of all commands, but naming is congruent to the command line interface. The only functionality that is not available at the command line is `datalad.api. Dataset`, DataLad's core Python data type. Just like any other command, it can be imported like this:
>
> ```
> >>> from datalad.api import Dataset
> ```
>
> or like this:
>
> ```
> >>> import datalad.api as dl
> >>> dl.Dataset()
> ```
>
> A `Dataset` is a class[129] that represents a DataLad dataset. In addition to the stand-alone commands, all of DataLad's functionality is also available via methods[130] of this class. Thus, these are two equally valid ways to create a new dataset with DataLad in Python:
>
> ```
> >>> from datalad.api import create, Dataset
> # create as a stand-alone command
> >>> create(path='scratch/test')
> [INFO   ] Creating a new annex repo at /home/me/scratch/test
> Out[3]: <Dataset path=/home/me/scratch/test>
>
> # create as a dataset method
> >>> ds = Dataset(path='scratch/test')
> >>> ds.create()
> [INFO   ] Creating a new annex repo at /home/me/scratch/test
> Out[3]: <Dataset path=/home/me/scratch/test>
> ```
>
> As shown above, the only required parameter for a Dataset is the `path` to its location, and this location may or may not exist yet.
>
> Stand-alone functions have a `dataset=` argument, corresponding to the `-d/ --dataset` option in their command-line equivalent. You can specify the `dataset=` argument with a path (string) to your dataset (such as `dataset='. '` for the current directory, or `dataset='path/to/ds'` to another location). Alternatively, you can pass a `Dataset` instance to it:

```
>>> from datalad.api import save, Dataset
# use save with dataset specified as a path
>>> save(dataset='path/to/dataset/')
# use save with dataset specified as a dataset instance
>>> ds = Dataset('path/to/dataset')
>>> save(dataset=ds, message="saving all modifications")
# use save as a dataset method (no dataset argument)
>>> ds.save(message="saving all modifications")
```

Use cases for DataLad's Python API

"Why should one use the Python API? Can we not do everything necessary via the command line already? Does Python add anything to this?" asks somebody.

It is completely up to on you and dependent on your preferred workflow whether you decide to use the command line or the Python API of DataLad for the majority of tasks. Both are valid ways to accomplish the same results. One advantage of using the Python API is the Dataset though: Given that the command line datalad command has a startup time (even when doing nothing) of ~200ms, this means that there is the potential for substantial speed-up when doing many calls to the API, and using a persistent Dataset object instance.

---

[128] http://docs.datalad.org/en/latest/modref.html
[129] https://docs.python.org/3/tutorial/classes.html
[130] https://docs.python.org/3/tutorial/classes.html#method-objects

> 💡 **M11.3 Creating an independent input dataset**
>
> If you acquire your own data for a data analysis, it will not magically exist as a DataLad dataset that you can simply install from somewhere – you'll have to turn it into a dataset yourself. Any directory with data that exists on your computer can be turned into a dataset with **datalad create --force** and a subsequent **datalad save -m "add data" .** to first create a dataset inside of an existing, non-empty directory, and subsequently save all of its contents into the history of the newly created dataset. And that's it already – it does not take anything more to create a stand-alone input dataset from existing data (apart from restraining yourself from modifying it afterwards).
> To create the `iris_data` dataset at https://github.com/datalad-handbook/iris_data we first created a DataLad dataset...
>
> ```
> # make sure to move outside of DataLad-101!
> $ cd ../
> $ datalad create iris_data
> [INFO] Creating a new annex repo at /home/me/dl-101/iris_data
> [INFO] scanning for unlocked files (this may take some time)
> create(ok): /home/me/dl-101/iris_data (dataset)
> ```
>
> and subsequently got the data from a publicly available GitHub Gist[133], a code snippet or other short standalone information (more on Gists here[134]), with a **datalad download-url** command:
>
> ```
> $ cd iris_data
> $ datalad download-url https://gist.githubusercontent.com/netj/8836201/
> ↪raw/6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv
> [INFO] Downloading 'https://gist.githubusercontent.com/netj/8836201/raw/
> ↪6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv' into '/home/me/dl-
> ↪101/iris_data/'
> download_url(ok): /home/me/dl-101/iris_data/iris.csv (file)
> add(ok): iris.csv (file)
> save(ok): . (dataset)
> action summary:
>   add (ok: 1)
>   download_url (ok: 1)
>   save (ok: 1)
> ```
>
> Finally, we *published* (more on this later in this section) the dataset to GITHUB.
> With this setup, the iris dataset (a single comma-separated (`.csv`) file) is downloaded, and, importantly, the dataset recorded *where* it was obtained from thanks to **datalad download-url**, thus complying to the second YODA principle. This way, upon installation of the dataset, DataLad knows where to obtain the file content from. You can **datalad clone** the iris dataset and find out with a `git annex whereis iris.csv` command.
>
> ---
> [133] https://gist.github.com/netj/8836201
> [134] https://docs.github.com/en/github/writing-on-github/editing-and-sharing-content-with-gists/creating-gists#about-gists

> 💡 **M11.7 On the looks and feels of this published dataset**
>
> Now that you have created and published such a YODA-compliant dataset, you are understandably excited how this dataset must look and feel for others. Therefore, you decide to install this dataset into a new location on your computer, just to get a feel for it. Replace the url in the **clone** command below with the path to your own midtermproject GitHub repository, or clone the "public" midterm_project repository that is available via the Handbook's GitHub organization at github.com/datalad-handbook/midterm_project[144]:
>
> ```
> $ cd ../../
> $ datalad clone "https://github.com/adswa/midtermproject.git"
> [INFO] Cloning dataset to Dataset(/home/me/dl-101/midtermproject)
> [INFO] Attempting to clone from https://github.com/adswa/midtermproject.git to /
> ↪home/me/dl-101/midtermproject
> [INFO] Start enumerating objects
> [INFO] Start counting objects
> [INFO] Start compressing objects
> [INFO] Start receiving objects
> [INFO] Start resolving deltas
> [INFO] Completed clone attempts for Dataset(/home/me/dl-101/midtermproject)
> [INFO] scanning for unlocked files (this may take some time)
> [INFO] Remote origin not usable by git-annex; setting annex-ignore
> install(ok): /home/me/dl-101/midtermproject (dataset)
> ```
>
> Let's start with the subdataset, and see whether we can retrieve the input iris.csv file. This should not be a problem, since its origin is recorded:
>
> ```
> $ cd midtermproject
> $ datalad get input/iris.csv
> [INFO] Cloning dataset to Dataset(/home/me/dl-101/midtermproject/input)
> [INFO] Attempting to clone from https://github.com/adswa/midtermproject.git/input␣
> ↪to /home/me/dl-101/midtermproject/input
> [INFO] Attempting to clone from https://github.com/adswa/midtermproject.git/input/.
> ↪git to /home/me/dl-101/midtermproject/input
> [INFO] Attempting to clone from https://github.com/datalad-handbook/iris_data.git␣
> ↪to /home/me/dl-101/midtermproject/input
> [INFO] Start enumerating objects
> [INFO] Start counting objects
> [INFO] Start compressing objects
> [INFO] Start receiving objects
> [INFO] Start resolving deltas
> [INFO] Completed clone attempts for Dataset(/home/me/dl-101/midtermproject/input)
> [INFO] scanning for unlocked files (this may take some time)
> [INFO] Remote origin not usable by git-annex; setting annex-ignore
> install(ok): /home/me/dl-101/midtermproject/input (dataset) [Installed subdataset␣
> ↪in order to get /home/me/dl-101/midtermproject/input/iris.csv]
> get(ok): input/iris.csv (file) [from web...]
> action summary:
>   get (ok: 1)
>   install (ok: 1)
> ```
>
> Nice, this worked well. The output files, however, can not be easily retrieved:

💡
```
$ datalad get prediction_report.csv pairwise_relationships.png
[ERROR] not available; (Note that these git remotes have annex-ignore set: origin)␣
↪[get(/home/me/dl-101/midtermproject/pairwise_relationships.png)]
get(error): pairwise_relationships.png (file) [not available; (Note that these git␣
↪remotes have annex-ignore set: origin)]
[ERROR] not available; (Note that these git remotes have annex-ignore set: origin)␣
↪[get(/home/me/dl-101/midtermproject/prediction_report.csv)]
get(error): prediction_report.csv (file) [not available; (Note that these git␣
↪remotes have annex-ignore set: origin)]
action summary:
  get (error: 2)
```

Why is that? This is the first detail of publishing datasets we will dive into. When publishing dataset content to GitHub with **datalad push**, it is the dataset's *history*, i.e., everything that is stored in Git, that is published. The file *content* of these particular files, though, is managed by GIT-ANNEX and not stored in Git, and thus only information about the file name and location is known to Git. Because GitHub does not host large data for free, annexed file content always needs to be deposited somewhere else (e.g., a web server) to make it accessible via **datalad get**. The chapter *Third party infrastructure* (page 177) will demonstrate how this can be done. For this dataset, it is not necessary to make the outputs available, though: Because all provenance on their creation was captured, we can simply recompute them with the **datalad rerun** command. If the tag was published we can simply rerun any **datalad run** command since this tag:

```
$ datalad rerun --since ready4analysis
```

But without the published tag, we can rerun the analysis by specifying its shasum:

```
$ datalad rerun d715890b36b9a089eedbb0c929f52e182e889735
[INFO] run commit d715890; (analyze iris data...)
[INFO] Making sure inputs are available (this may take some time)
[WARNING] no content present; cannot unlock [unlock(/home/me/dl-101/midtermproject/
↪pairwise_relationships.png)]
[WARNING] no content present; cannot unlock [unlock(/home/me/dl-101/midtermproject/
↪prediction_report.csv)]
remove(ok): pairwise_relationships.png
remove(ok): prediction_report.csv
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): pairwise_relationships.png (file)
add(ok): prediction_report.csv (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 3)
  remove (ok: 2)
  save (notneeded: 1, ok: 1)
```

Hooray, your analysis was reproduced! You happily note that rerunning your analysis was incredibly easy – it would not even be necessary to have any knowledge about the analysis at all to reproduce it! With this, you realize again how letting DataLad take care of linking input, output, and code can make your life and others' lives so much easier. Applying the YODA principles to your data analysis was very beneficial indeed. Proud of your midterm project you can not wait to use those principles the next time again.

---
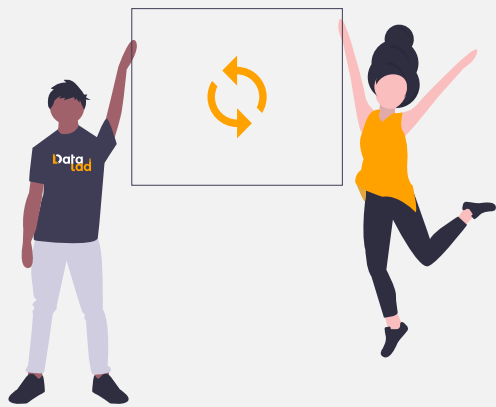
[144] https://github.com/datalad-handbook/midterm_project

ONE STEP FURTHER

12.1 More on Dataset nesting

You may have noticed how working in the subdataset felt as if you would be working in an independent dataset – there was no information or influence at all from the top-level `DataLad-101` superdataset, and you build up a completely stand-alone history:

```
$ git log --oneline
0687926 Provide project description
f76acca [DATALAD RUNCMD] analyze iris data with classification analysis
dd36472 add script for kNN classification and plotting
01447d9 [DATALAD] modified subdataset properties
2465103 [DATALAD] Recorded changes
261ea45 Apply YODA dataset setup
50e40b2 [DATALAD] new dataset
```

In principle, this is no news to you. From section *Dataset nesting* (page 53) and the YODA principles you already know that nesting allows for a modular re-use of any other DataLad dataset, and that this re-use is possible and simple precisely because all of the information is kept within a (sub)dataset.

What is new now, however, is that you applied changes to the dataset. While you already explored the looks and feels of the `longnow` subdataset in previous sections, you now *modified* the contents of the `midterm_project` subdataset. How does this influence the superdataset, and how does this look like in the superdataset's history? You know from section *Dataset nesting* (page 53) that the superdataset only stores the *state* of the subdataset. Upon creation of the dataset, the very first, initial state of the subdataset was thus recorded in the superdataset. But

now, after you finished your project, your subdataset evolved. Let's query the superdataset what it thinks about this.

```
# move into the superdataset
$ cd ../
$ datalad status
 modified: midterm_project (dataset)
```

From the superdataset's perspective, the subdataset appears as being "modified". Note how it is not individual files that show up as "modified", but indeed the complete subdataset as a single entity.

What this shows you is that the modifications of the subdataset you performed are not automatically recorded to the superdataset. This makes sense – after all it should be up to you to decide whether you want record something or not –, but it is worth repeating: If you modify a subdataset, you will need to save this *in the superdataset* in order to have a clean superdataset status.

This point in time in DataLad-101 is a convenient moment to dive a bit deeper into the functions of the **datalad status** command. If you are interested in this, checkout the *dedicated Findoutmore* (page 173).

Let's save the modification of the subdataset into the history of the superdataset. For this, to avoid confusion, you can specify explicitly to which dataset you want to save a modification. -d . specifies the current dataset, i.e., DataLad-101, as the dataset to save to:

```
$ datalad save -d . -m "finished my midterm project" midterm_project
add(ok): midterm_project (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

> 💡 **M12.2 More on how save can operate on nested datasets**
>
> In a superdataset with subdatasets, **datalad save** by default tries to figure out on its own which dataset's history of all available datasets a **save** should be written to. However, it can reduce confusion or allow specific operations to be very explicit in the command call and tell DataLad where to save what kind of modifications to.
> If you want to save the current state of the subdataset into the superdataset (as necessary here), start a save from the superdataset and have the -d/--dataset option point to its root:
>
> ```
> # in the root of the superds
> $ datalad save -d . -m "update subdataset"
> ```
>
> If you are in the superdataset, and you want to save an unsaved modification in a subdataset to the *subdatasets* history, let -d/--dataset point to the subdataset:
>
> ```
> # in the superds
> $ datalad save -d path/to/subds -m "modified XY"
> ```
>
> The recursive option allows you to save any content underneath the specified directory, and recurse into any potential subdatasets:
>
> ```
> $ datalad save . --recursive
> ```

Let's check which subproject commit is now recorded in the superdataset:

```
$ git log -p -n 1
commit 4d216b85af27e7fdabd0d2a62e9d8e8273064cab
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:23:05 2021 +0200

    finished my midterm project

diff --git a/midterm_project b/midterm_project
index 261ea45..0687926 160000
--- a/midterm_project
+++ b/midterm_project
@@ -1 +1 @@
-Subproject commit 261ea45212b615fc7aca52d6f3509a731d666001
+Subproject commit 0687926665d919ee11ae41cf6be1c16d0a9d958c
```

As you can see in the log entry, the subproject commit changed from the first commit hash in the subdataset history to the most recent one. With this change, therefore, your superdataset tracks the most recent version of the `midterm_project` dataset, and your dataset's status is clean again.

12.2 Computational reproducibility with software containers

Just after submitting your midterm data analysis project, you get together with your friends. "I'm curious: So what kind of analyses did y'all carry out?" you ask. The variety of methods and datasets the others used is huge, and one analysis interests you in particular. Later that day, you decide to install this particular analysis dataset to learn more about the methods used in there. However, when you `re-run` your friends analysis script, it throws an error. Hastily, you call her – maybe she can quickly fix her script and resubmit the project with only minor delays. "I don't know what you mean", you hear in return. "On my machine, everything works fine!"

On its own, DataLad datasets can contain almost anything that is relevant to ensure reproducibility: Data, code, human-readable analysis descriptions (e.g., `README.md` files), provenance on the origin of all files obtained from elsewhere, and machine-readable records that link generated outputs to the commands, scripts, and data they were created from.

This however may not be sufficient to ensure that an analysis *reproduces* (i.e., produces the same or highly similar results), let alone *works* on a computer different than the one it was initially composed on. This is because the analysis does not only depend on data and code, but also the *software environment* that it is conducted in.

A lack of information about the operating system of the computer, the precise versions of installed software, or their configurations may make it impossible to replicate your analysis on a different machine, or even on your own machine once a new software update is installed. Therefore, it is important to communicate all details about the computational environment for an analysis as thoroughly as possible. Luckily, DataLad provides an extension that can link computational environments to datasets, the datalad containers[156] extension[169].

This section will give a quick overview on what containers are and demonstrate how `datalad-containers` helps to capture full provenance of an analysis by linking containers to datasets and analyses.

---

[156] http://docs.datalad.org/projects/container/en/latest/
[169] To read more about DataLad's extensions, see section *DataLad extensions* (page 285).

Note: In order to run the commands in this section, the *datalad-containers* extension needs to be installed separately, simply with *pip install datalad-container*.

Containers

To put it simple, computational containers are cut-down virtual machines that allow you to package all software libraries and their dependencies (all in the precise version your analysis requires) into a bundle you can share with others. On your own and other's machines, the container constitutes a secluded software environment that

- contains the exact software environment that you specified, ready to run analyses in
- does not effect any software outside of the container

Unlike virtual machines, software containers do not have their own operating system. Instead, they use basic services of the underlying operating system of the computer they run on (in a read-only fashion). This makes them lightweight and portable. By sharing software environments with containers, others (and also yourself) have easy access to the correct software without the need to modify the software environment of the machine the container runs on. Thus, containers are ideal to encapsulate the software environment and share it together with the analysis code and data to ensure computational reproducibility of your analyses, or to create a suitable software environment on a computer that you do not have permissions to deploy software on.

There are a number of different tools to create and use containers, with Docker[157] being one of the most well-known of them. While being a powerful tool, it is only rarely used on high performance computing (HPC) infrastructure[170]. An alternative is Singularity[158]. Both of these tools share core terminology:

Recipe A text file template that lists all required components of the computational environment. It is made by a human user.

Image This is *built* from the recipe file. It is a static filesystem inside a file, populated with the software specified in the recipe, and some initial configuration.

Container A running instance of an Image that you can actually use for your computations. If you want to create and run your own software container, you start by writing a recipe file and build an Image from it. Alternatively, you can can also *pull* an Image built from a publicly shared recipe from the *Hub* of the tool you are using.

Hub A storage resource to share and consume images. Among the most popular registries are Singularity-Hub[159] and Docker-Hub[160]. Both are optional, additional services not required to use software containers, but a convenient way to share recipes and have imaged built from them by a service (instead of building them manually and locally). Another large container registry is Amazon ECR[161] which hosts Docker Images.

---

[157] https://www.docker.com/

[170] The main reason why Docker is not deployed on HPC systems is because it grants users "superuser privileges[171]". On multi-user systems such as HPC, users should not have those privileges, as it would enable them to tamper with other's or shared data and resources, posing a severe security threat.

[171] https://en.wikipedia.org/wiki/Superuser

[158] https://sylabs.io/docs/

[159] https://singularity-hub.org/

[160] https://hub.docker.com/

[161] https://aws.amazon.com/ecr/

Note that as of now, the `datalad-containers` extension supports Singularity and Docker images. Singularity furthermore is compatible with Docker – you can use Docker Images as a basis for Singularity Images, or run Docker Images with Singularity (even without having Docker installed).

> ⚠️ **Additional requirement: Singularity**
>
> In order to use Singularity containers (and thus `datalad containers`), you have to install[162] the software singularity.
>
> ---
> [162] https://sylabs.io/guides/3.0/user-guide/installation.html

Using `datalad containers`

One core feature of the `datalad containers` extension is that it registers computational containers to a dataset. This is done with the **`datalad containers-add`** command. Once a container is registered, arbitrary commands can be executed inside of it, i.e., in the precise software environment the container encapsulates. All it needs for this it to swap the **`datalad run`** command introduced in section *Keeping track* (page 59) with the **`datalad containers-run`** command.

Let's see this in action for the `midterm_analysis` dataset by rerunning the analysis you did for the midterm project within a Singularity container. We start by registering a container to the dataset. For this, we will pull an Image from Singularity hub. This Image was made for the handbook, and it contains the relevant Python setup for the analysis. Its recipe lives in the handbook's resources repository[163], and the Image is built from the recipe via Singularity hub. If you're curious how to create a Singularity Image, the hidden section below has some pointers:

> 💡 **M12.3 How to make a Singularity Image**
>
> Singularity containers are build from Image files, often called "recipes", that hold a "definition" of the software container and its contents and components. The singularity documentation[164] has its own tutorial on how to build such Images from scratch. An alternative to writing the Image file by hand is to use Neurodocker[165]. This command-line program can help you generate custom Singularity recipes (and also `Dockerfiles`, from which Docker Images are build). A wonderful tutorial on how to use Neurodocker is this introduction[166] by Michael Notter.
> Once a recipe exists, the command
>
> ```
> sudo singularity build <NAME> <RECIPE>
> ```
>
> will build a container (called `<NAME>`) from the recipe. Note that this command requires `root` privileges ("sudo"). You can build the container on any machine, though, not necessarily the one that is later supposed to actually run the analysis, e.g., your own laptop versus a compute cluster. Alternatively, Singularity Hub[167] integrates with Github and builds containers from Images pushed to repositories on Github. The docs[168] give you a set of instructions on how to do this.
>
> ---
> [164] https://sylabs.io/guides/3.4/user-guide/build_a_container.html
> [165] https://github.com/ReproNim/neurodocker
> [166] https://miykael.github.io/nipype_tutorial/notebooks/introduction_neurodocker.html
> [167] https://singularity-hub.org/
> [168] https://singularityhub.github.io/singularityhub-docs/

---
[163] https://github.com/datalad-handbook/resources

The **datalad containers-add** command takes an arbitrary name to give to the container, and a path or url to a container Image:

```
# we are in the midterm_project subdataset
$ datalad containers-add midterm-software --url shub://adswa/resources:2
[INFO] Initiating special remote datalad
add(ok): .datalad/config (file)
save(ok): . (dataset)
containers_add(ok): /home/me/dl-101/DataLad-101/midterm_project/.datalad/environments/
↪midterm-software/image (file)
action summary:
  add (ok: 1)
  containers_add (ok: 1)
  save (ok: 1)
```

> **M12.4 How do I add an Image from Dockerhub, Amazon ECR, or a local container?**
>
> Should the Image you want to use lie on Dockerhub, specify the `--url` option prefixed with docker:// or dhub:// instead of shub:// like this:
>
> ```
> datalad containers-add midterm-software --url docker://adswa/resources:2
> ```
>
> If your Image exists on Amazon ECR, use a dhub:// prefix followed by the AWS ECR URL as in
>
> ```
> datalad containers-add --url dhub://12345678.dkr.ecr.us-west-2.amazonaws.com/maze-
> ↪code/data-import:latest data-import
> ```
>
> If you want to add a container that exists locally, specify the path to it like this:
>
> ```
> datalad containers-add midterm-software --url path/to/container
> ```

This command downloaded the container from Singularity Hub, added it to the `midterm_project` dataset, and recorded basic information on the container under its name "midterm-software" in the dataset's configuration at `.datalad/config`. You can find out more about them in a dedicated *find-out-more on these additional configurations* (page 175).

Now that we have a complete computational environment linked to the `midterm_project` dataset, we can execute commands in this environment. Let us for example try to repeat the **datalad run** command from the section *YODA-compliant data analysis projects* (page 143) as a **datalad containers-run** command.

The previous `run` command looked like this:

```
$ datalad run -m "analyze iris data with classification analysis" \
  --input "input/iris.csv" \
  --output "prediction_report.csv" \
  --output "pairwise_relationships.png" \
  "python3 code/script.py"
```

How would it look like as a `containers-run` command?

```
$ datalad containers-run -m "rerun analysis in container" \
  --container-name midterm-software \
  --input "input/iris.csv" \
  --output "prediction_report.csv" \
  --output "pairwise_relationships.png" \
```

<div align="right">(continues on next page)</div>

```
  "python3 code/script.py"
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): pairwise_relationships.png (file)
unlock(ok): prediction_report.csv (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): pairwise_relationships.png (file)
add(ok): prediction_report.csv (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 4)
  save (notneeded: 1, ok: 1)
  unlock (ok: 2)
```

Almost exactly like a **datalad run** command! The only additional parameter is container-name.
At this point, though, the --container-name flag is even *optional* because there is only a single
container registered to the dataset. But if your dataset contains more than one container you
will *need* to specify the name of the container you want to use in your command. The complete
command's structure looks like this:

```
$ datalad containers-run --name <containername> [-m ...] [--input ...] [--output ...]
↪<COMMAND>
```

> **M12.6 How can I list available containers or remove them?**
>
> The command **datalad containers-list** will list all containers in the current dataset:
>
> ```
> $ datalad containers-list
> midterm-software -> .datalad/environments/midterm-software/image
> ```
>
> The command **datalad containers-remove** will remove a container from the dataset, if
> there exists a container with name given to the command. Note that this will remove not
> only the Image from the dataset, but also the configuration for it in .datalad/config.

Here is how the history entry looks like:

```
$ git log -p -n 1
commit e231e465180918f572c1f2954302b7b958f82c52
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:24:53 2021 +0200

    [DATALAD RUNCMD] rerun analysis in container

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "singularity exec .datalad/environments/midterm-software/image python3 code/
↪script.py",
     "dsid": "fea0e9c7-7932-41d5-8d57-97ac9ac6755a",
     "exit": 0,
     "extra_inputs": [
      ".datalad/environments/midterm-software/image"
     ],
     "inputs": [
```

```
      "input/iris.csv"
     ],
     "outputs": [
      "prediction_report.csv",
      "pairwise_relationships.png"
     ],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^

diff --git a/pairwise_relationships.png b/pairwise_relationships.png
index c3e4d53..963d5a8 120000
--- a/pairwise_relationships.png
+++ b/pairwise_relationships.png
@@ -1 +1 @@
-.git/annex/objects/09/M8/MD5E-s260804--3bf42e2d4c2a88ec0ddc5b16c4151c68.png/MD5E-s260804-
↪-3bf42e2d4c2a88ec0ddc5b16c4151c68.png
\ No newline at end of file
+.git/annex/objects/q1/gp/MD5E-s261062--025dc493ec2da6f9f79eb1ce8512cbec.png/MD5E-s261062-
↪-025dc493ec2da6f9f79eb1ce8512cbec.png
\ No newline at end of file
```

If you would **rerun** this commit, it would be re-executed in the software container registered to the dataset. If you would share the dataset with a friend and they would **rerun** this commit, the Image would first be obtained from its registered url, and thus your friend can obtain the correct execution environment automatically.

Note that because this new **containers-run** command modified the midterm_project subdirectory, we need to also save the most recent state of the subdataset to the superdataset DataLad-101.

```
$ cd ../
$ datalad status
 modified: midterm_project (dataset)

$ datalad save -d . -m "add container and execute analysis within container" midterm_
↪project
add(ok): midterm_project (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Software containers, the datalad-containers extension, and DataLad thus work well together to make your analysis completely reproducible – by not only linking code, data, and outputs, but also the software environment of an analysis. And this does not only benefit your future self, but also whomever you share your dataset with, as the information about the container is shared together with the dataset. How cool is that?

If you are interested in more, you can read about another example of **datalad containers-run** in the usecase *An automatically and computationally reproducible neuroimaging analysis from scratch* (page 420).

12.3 Summary

The last two sections have first of all extended your knowledge on dataset nesting:

- When subdatasets are created or installed, they are registered to the superdataset in their current version state (as identified by their most recent commit's hash). For a freshly created subdatasets, the most recent commit is at the same time its first commit.

- Once the subdataset evolves, the superdataset recognizes this as a `modification` of the subdatasets version state. If you want to record this, you need to **save** it in the super-dataset:

  ```
  $ datalad save -m "a short summary of changes in subds" <path to subds>
  ```

But more than nesting concepts, they have also extended your knowledge on reproducible analyses with **datalad run** and you have experienced for yourself why and how software containers can go hand-in-hand with DataLad:

- A software container encapsulates a complete software environment, independent from the environment of the computer it runs on. This allows you to create or use secluded software and also share it together with your analysis to ensure computational reproducibility. The DataLad extension datalad containers[172] can make this possible.

- The command **datalad containers-add** registers an Image from a path or url to your dataset.

- If you use **datalad containers-run** instead of **datalad run**, you can reproducibly execute a command of your choice *within* the software environment.

- A **datalad rerun** of a commit produced with **datalad containers-run** will re-execute the command in the same software environment.

Now what can I do with it?

For one, you will not be surprised if you ever see a subdataset being shown as `modified` by **datalad status**: You now know that if a subdataset evolves, it's most recent state needs to be explicitly saved to the superdatasets history.

On a different matter, you are now able to capture and share analysis provenance that includes the relevant software environment. This does not only make your analyses projects automatically reproducible, but automatically *computationally* reproducible - you can make sure that your analyses runs on any computer with Singularity, regardless of the software environment on this computer. Even if you are unsure how you can wrap up an environment into a software container Image at this point, you could make use of hundreds of publicly available Images on Singularity-Hub[173] and Docker-Hub[174].

With this, you have also gotten a first glimpse into an extension of DataLad: A Python module you can install with Python package managers such as `pip` that extends DataLad's functionality.

---

[172] http://docs.datalad.org/projects/container/en/latest/
[173] https://singularity-hub.org/
[174] https://hub.docker.com/

> **M12.1 More on datalad status**
>
> First of all, let's start with a quick overview of the different content *types* and content *states* various **datalad status** commands in the course of DataLad-101 have shown up to this point:
>
> You have seen the following *content types*:
>
> - file, e.g., notes.txt: any file (or symlink that is a placeholder to an annexed file)
> - directory, e.g., books: any directory that does not qualify for the dataset type
> - symlink, e.g., the .jgp that was manually unlocked in section *Input and output* (page 70): any symlink that is not used as a placeholder for an annexed file
> - dataset, e.g., the midterm_project: any top-level dataset, or any subdataset that is properly registered in the superdataset
>
> And you have seen the following *content states*: modified and untracked. The section *Miscellaneous file system operations* (page 224) will show you many instances of deleted content state as well.
>
> But beyond understanding the report of **datalad status**, there is also additional functionality: **datalad status** can handle status reports for a whole hierarchy of datasets, and it can report on a subset of the content across any number of datasets in this hierarchy by providing selected paths. This is useful as soon as datasets become more complex and contain subdatasets with changing contents.
>
> When performed without any arguments, **datalad status** will report the state of the current dataset. However, you can specify a path to any sub- or superdataset with the --dataset option.
>
> In order to demonstrate this a bit better, we will make sure that not only the state of the subdataset *within* the superdataset is modified, but also that the subdataset contains a modification. For this, let's add an empty text file into the midterm_project subdataset:
>
> ```
> $ touch midterm_project/an_empty_file
> ```
>
> If you are in the root of DataLad-101, but interested in the status *within* the subdataset, simply provide a path (relative to your current location) to the command:
>
> ```
> $ datalad status midterm_project
> untracked: midterm_project/an_empty_file (file)
> ```
>
> Alternatively, to achieve the same, specify the superdataset as the --dataset and provide a path to the subdataset *with a trailing path separator* like this:
>
> ```
> $ datalad status -d . midterm_project/
> untracked: midterm_project/an_empty_file (file)
> ```
>
> Note that both of these commands return only the untracked file and not not the modified subdataset because we're explicitly querying only the subdataset for its status. If you however, as done outside of this hidden section, you want to know about the subdataset record in the superdataset without causing a status query for the state *within* the subdataset itself, you can also provide an explicit path to the dataset (without a trailing path separator). This can be used to specify a specific subdataset in the case of a dataset with many subdatasets:
>
> ```
> $ datalad status -d . midterm_project
>  modified: midterm_project (dataset)
> ```
>
> But if you are interested in both the state within the subdataset, and the state of the subdataset within the superdataset, you can combine the two paths:

```
$ datalad status -d . midterm_project midterm_project/
 modified: midterm_project (dataset)
untracked: midterm_project/an_empty_file (file)
```

Finally, if these subtle differences in the paths are not easy to memorize, the `-r`/
`--recursive` option will also report you both status aspects:

```
$ datalad status --recursive
 modified: midterm_project (dataset)
untracked: midterm_project/an_empty_file (file)
```

This still was not all of the available functionality of the **datalad status** command. You
could for example adjust whether and how untracked dataset content should be reported
with the `--untracked` option, or get additional information from annexed content with
the `--annex` option. To get a complete overview on what you could do, check out the
technical documentation of **datalad status** here[155].

Before we leave this hidden section, lets undo the modification of the subdataset by
removing the untracked file:

```
$ rm midterm_project/an_empty_file
$ datalad status --recursive
 modified: midterm_project (dataset)
```

---

[155] http://docs.datalad.org/en/latest/generated/man/datalad-status.html

> **M12.5 What changes in .datalad/config when one adds a container?**
>
> ```
> $ cat .datalad/config
> [datalad "dataset"]
>         id = fea0e9c7-7932-41d5-8d57-97ac9ac6755a
> [datalad "containers.midterm-software"]
>         image = .datalad/environments/midterm-software/image
>         cmdexec = singularity exec {img} {cmd}
> ```
>
> This recorded the Image's origin on Singularity-Hub, the location of the Image in the dataset under .datalad/environments/<NAME>/image, and it specifies the way in which the container should be used: The line
>
> ```
> cmdexec = singularity exec {img} {cmd}
> ```
>
> can be read as: "If this container is used, take the cmd (what you wrap in a **datalad containers-run** command) and plug it into a **singularity exec** command. The mode of calling Singularity, namely exec, means that the command will be executed inside of the container.
>
> You can configure this call format by modifying it in the config file, or calling **datalad containers-add** with the option --call-fmt <alternative format>. This can be useful to, for example, automatically bind-mount the current working directory in the container. In the alternative call format, the placeholders {img}, {cmd}, and {img_dspath} (a relative path to the dataset containing the image) are available. In all other cases with variables that use curly brackets, you need to escape them with another curly bracket. Here is an example call format that bind-mounts the current working directory (and thus the dataset) automatically:
>
> ```
> datalad containers-add --call-fmt 'singularity exec -B {{pwd}} --cleanenv {img}
> ↪{cmd}'
> ```
>
> Note that the Image is saved under .datalad/environments and the configuration is done in .datalad/config – as these files are version controlled and shared with together with a dataset, your software container and the information where it can be re-obtained from are linked to your dataset.
>
> This is how the containers-add command is recorded in your history:

```
$ git log -n 1 -p
commit 4abf32c5ab84aca1c6a1b54da71a0b14e395bceb
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:24:46 2021 +0200

    [DATALAD] Configure containerized environment 'midterm-software'

diff --git a/.datalad/config b/.datalad/config
index f2270b6..efefa2d 100644
--- a/.datalad/config
+++ b/.datalad/config
@@ -1,2 +1,5 @@
 [datalad "dataset"]
        id = fea0e9c7-7932-41d5-8d57-97ac9ac6755a
+[datalad "containers.midterm-software"]
+       image = .datalad/environments/midterm-software/image
+       cmdexec = singularity exec {img} {cmd}
diff --git a/.datalad/environments/midterm-software/image b/.datalad/environments/
↪midterm-software/image
new file mode 120000
index 0000000..75c8b41
--- /dev/null
+++ b/.datalad/environments/midterm-software/image
@@ -0,0 +1 @@
+../../../.git/annex/objects/F1/K3/MD5E-s230694943--
↪944b0300fab145c7ebbd86078498b393/MD5E-s230694943--944b0300fab145c7ebbd86078498b393
\ No newline at end of file
```

THIRD PARTY INFRASTRUCTURE
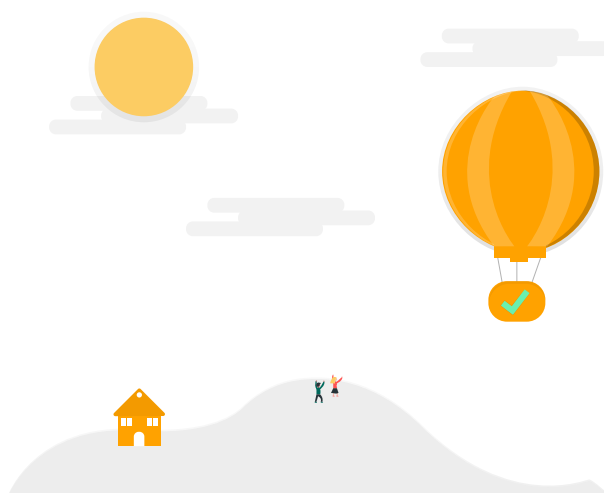
13.1 Beyond shared infrastructure

Data sharing potentially involves a number of different elements:
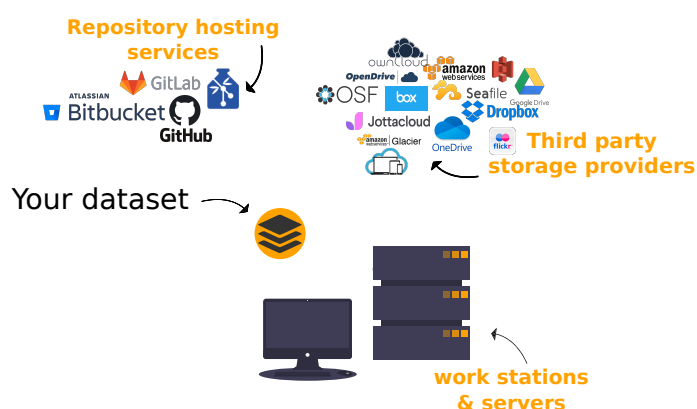
Fig. 13.1: An overview of all elements potentially included in a publication workflow.

Users on a common, shared computational infrastructure such as an SSH SERVER can share datasets via simple installations with paths, without any involvement of third party storage providers or repository hosting services:

Clone a local path (on the same machine)

datalad clone <path/to/dataset>

datalad get retrieves annexed files automatically from local path

Clone from a server/cluster

datalad clone \
    <user@server:/path/to/dataset>
datalad get retrieves annexed files automatically from server
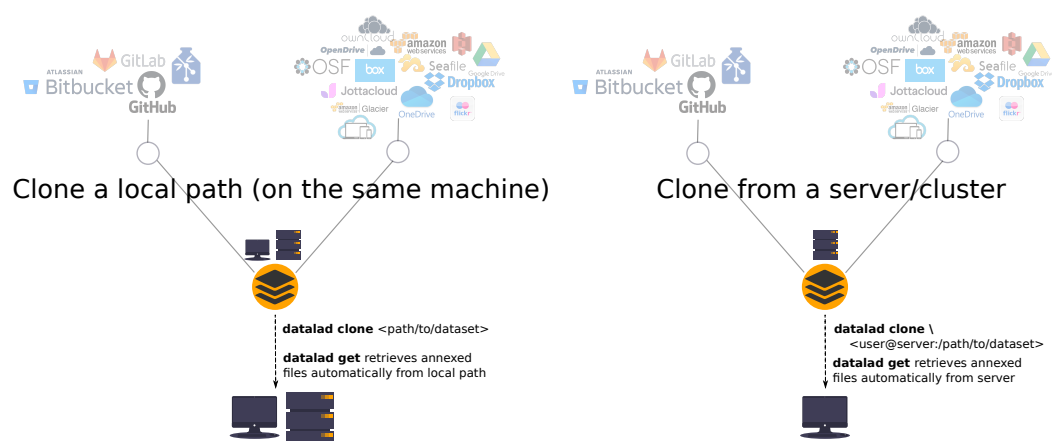
But at some point in a dataset's life, you may want to share it with people that can't access the computer or server your dataset lives on, store it on other infrastructure to save diskspace, or create a backup. When this happens, you will want to publish your dataset to repository hosting services (for example GITHUB, GITLAB, or GIN) and/or third party storage providers (such as Dropbox[175], Google[176], Amazon S3 buckets[177], the Open Science Framework (OSF)[178], and many others).

This chapter tackles different aspects of dataset publishing. The remainder of this section talks about general aspects of dataset publishing, and illustrates the idea of using third party services as SPECIAL REMOTEs from which annexed file contents can be retrieved via **datalad get**.

The upcoming section *Walk-through: Dataset hosting on GIN* (page 208) shows you one of the most easy ways to publish your dataset publicly or for selected collaborators and friends. If you don't want to dive in to all the details on dataset sharing, it is safe to directly skip ahead to this section, and have your dataset published in only a few minutes.

Other sections in this chapter will showcase a variety of ways to publish datasets and their contents to different services: The section *Publishing datasets to Git repository hosting* (page 184) demonstrates how to publish datasets to any kind of Git repository hosting service. The sections *Walk-through: Amazon S3 as a special remote* (page 197) and *Walk-through: Dropbox as a special remote* (page 191) are concrete examples of sharing datasets publicly or with selected others via different cloud services. The section *Walk-through: Git LFS as a special remote on GitHub* (page 207) talks about using the centralized, for-pay service Git LFS[179] for sharing dataset content on GitHub, and the section *Built-in data export* (page 215) shows built-in dataset export to services such as figshare.com[180]. If you want a walk-through for a different service, or if you maybe even want to share your own walk-through, please get in touch[181].

> ⚠️ **There can never be "too much" documentation**
>
> If you plan to share your own datasets with people that are unfamiliar with DataLad, it may be helpful to give a short explanation of what a DataLad dataset is and what it can do. For this, you can use a ready-made text block that the handbook provides. To find this textblock, go to *How can I help others get started with a shared dataset?* (page 482). Alternative, run **datalad add-readme**.

---

[175] https://dropbox.com
[176] https://google.com
[177] https://aws.amazon.com/s3/?nc1=h_ls
[178] https://osf.io/
[179] https://git-lfs.github.com/
[180] https://figshare.com/
[181] https://github.com/datalad-handbook/book/issues/new

Leveraging third party infrastructure

There are several ways to make datasets available for others:

- You can **publish your dataset to a repository with annex support** such as GIN or the Open Science Framework (OSF)[182,208]. This is the easiest way to share datasets and all their contents. Read on in the section *Walk-through: Dataset hosting on GIN* (page 208) or consult the tutorials of the datalad-osf extension[183] to learn how to do this.

- You can **publish your dataset to a repository hosting service**, and **configure an external resource that stores your annexed data**. Such a resource can be a private web server, but also a third party services cloud storage such as Dropbox[184], Google[185], Amazon S3 buckets[186], Box.com[187], owncloud[188], sciebo[189], or many more.

- You can **export your dataset statically** as a snapshot to a service such as Figshare[190] or the Open Science Framework (OSF)[191][208].

- You can **publish your dataset to a repository hosting service** and ensure that all dataset contents are either available from pre-existing public sources or can be recomputed from a RUN RECORD.

Dataset contents and third party services influence sharing

Because DataLad datasets are GIT repositories, it is possible to **push** datasets to any Git repository hosting service, such as GITHUB, GITLAB, GIN, BITBUCKET, Gogs[192], or Gitea[193]. You have already done this in section *YODA-compliant data analysis projects* (page 143) when you shared your midterm_project dataset via GITHUB.

However, most Git repository hosting services do not support hosting the file content of the files managed by GIT-ANNEX. For example, the the results of the analysis in section *YODA-compliant data analysis projects* (page 143), pairwise_comparisons.png and prediction_report.csv, were not published to GitHub: There was meta data about their file availability, but if a friend cloned this dataset and ran a **datalad get** command, content retrieval would fail because their only known location is your private computer to which only you have access. Instead, they would need to be recomputed from the RUN RECORD in the dataset.

When you are sharing DataLad datasets with other people or third party services, an important distinction thus lies in *annexed* versus *not-annexed* content, i.e., files that stored in your dataset's ANNEX versus files that are committed into GIT. The third-party service of your choice may have support for both annexed and non-annexed files, or only one them.

---

[182] https://osf.io/
[208] Requires the datalad-osf extension[209].
[209] http://docs.datalad.org/projects/osf/en/latest/index.html
[183] http://docs.datalad.org/projects/osf/en/latest/index.html
[184] https://dropbox.com
[185] https://google.com
[186] https://aws.amazon.com/s3/?nc1=h_ls
[187] https://www.box.com/en-gb/home
[188] https://owncloud.com
[189] https://hochschulcloud.nrw
[190] https://figshare.com/
[191] https://osf.io/
[192] https://gogs.io/
[193] https://gitea.io/en-us/
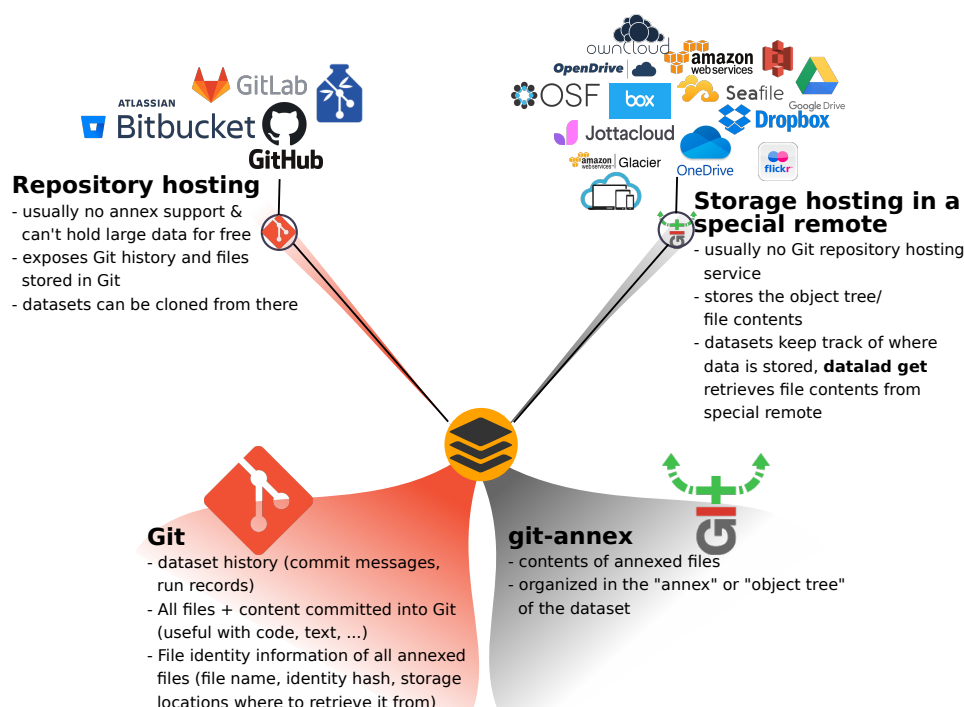
Fig. 13.2: Schematic difference between the Git and git-annex aspect of your dataset, and where each part *usually* gets published to.

The common case: Repository hosting without annex support and special remotes

Because DataLad datasets are GIT repositories, it is possible to **push** datasets to any Git repository hosting service, such as GITHUB, GITLAB, GIN, BITBUCKET, Gogs[194], or Gitea[195]. But while anything that is managed by Git is accessible in repository hosting services, they usually don't support storing annexed data[210].

When you want to publish a dataset to a Git repository hosting service to allow others to easily find and clone it, but you also want others to be able to retrieve annexed files in this dataset via `datalad get`, annexed contents need to be pushed to additional storage hosting services. The hosting services can be all kinds of private, institutional, or commercial services, and their location will be registered in the dataset under the concept of a SPECIAL REMOTE.

> **M13.1 What is a special remote**
>
> A special-remote is an extension to Git's concept of remotes, and can enable GIT-ANNEX to transfer data from and possibly to places that are not Git repositories (e.g., cloud services or external machines such as an HPC system). For example, an *s3* special remote uploads and downloads content to AWS S3, a *web* special remote downloads files from the web, and *datalad-archive* extracts files from the annexed archives, etc. Don't envision a special-remote as merely a physical place or location – a special-remote is a protocol that defines the underlying transport of your files to and/or from a specific location.

---

[194] https://gogs.io/

[195] https://gitea.io/en-us/

[210] In addition to not storing annexed data, most Git repository hosting services also have a size limit for files kept in Git. So while you could *theoretically* commit a sizable file into Git, this would not only negatively impact the performance of your dataset as Git doesn't handle large files well, but it would also prevent your dataset to be published to a Git repository hosting service like GitHub[211].

[211] https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github

To register a special remote in your dataset and use it for file storage, you need to configure the service of your choice and *publish* the annexed contents to it. Afterwards, the published dataset (e.g., via GITHUB or GITLAB) stores the information about where to obtain annexed file contents from such that `datalad get` works. Once you have configured the service of your choice, you can push your datasets Git history to the repository hosting service and the annexed contents to the special remote. But DataLad also makes it easy to push these different dataset contents exactly where they need to be automatically via a PUBLICATION DEPENDENCY.

Exemplary walk-throughs for Dropbox[196], Amazon S3 buckets[197], and Git LFS[198] can be found in the upcoming sections in this chapter. But the general workflow looks as follows:

From your perspective (as someone who wants to share data), you will need to

- (potentially) install/setup the relevant *special-remote*,

- create a dataset sibling on GitHub/GitLab/... for others to install from

- set up a *publication dependency* between repository hosting and special remote , so that annexed contents are automatically pushed to the special remote when ever you update the sibling on the Git repository hosting site

- publish your dataset

This gives you the freedom to decide where your data lives and who can have access to it. Once this set up is complete, updating and accessing a published dataset and its data is almost as easy as if it would lie on your own machine.

From the perspective of a consumer (as someone who wants to obtain your dataset), they will need to

- (potentially) install the relevant *special-remote* (dependent on the third-party service you chose) and

- perform the standard `datalad clone` and `datalad get` commands as necessary.

Thus, from a collaborator's perspective, with the exception of potentially installing/setting up the relevant *special-remote*, obtaining your dataset and its data is as easy as with any public DataLad dataset. While you have to invest some setup effort in the beginning, once this is done, the workflows of yours and others are the same that you are already very familiar with.

datalad clone <URL>

datalad get retrieves annexed files from the original location or the special remote. May need configuration by user or dataset provider.

Clone from a remote URL

---

[196] https://dropbox.com
[197] https://aws.amazon.com/s3/?nc1=h_ls
[198] https://github.com/git-lfs/git-lfs
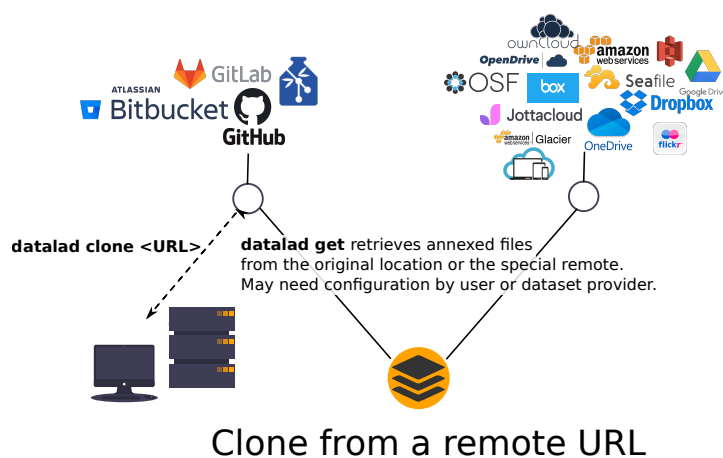
If you are interested in learning how to set up different services as special remotes, you can take a look at the sections *Walk-through: Amazon S3 as a special remote* (page 197), *Walk-through: Dropbox as a special remote* (page 191) or *Walk-through: Git LFS as a special remote on GitHub* (page 207) for concrete examples with DataLad datasets, and the general section *Publishing datasets to Git repository hosting* (page 184) on setting up dataset siblings. In addition, there are step-by-step walk-throughs in the documentation of git-annex for services such as S3[199], Google Cloud Storage[200], Box.com[201], Amazon Glacier[202], OwnCloud[203], and many more. Here is the complete list: git-annex.branchable.com/special_remotes[204].

The easy case: Repository hosting with annex support

There are a few Git repository hosting services with support for annexed contents. One of them is GIN. What makes them extremely convenient is that there is no need to configure a special remote – creating a SIBLING and running `datalad push` is enough.
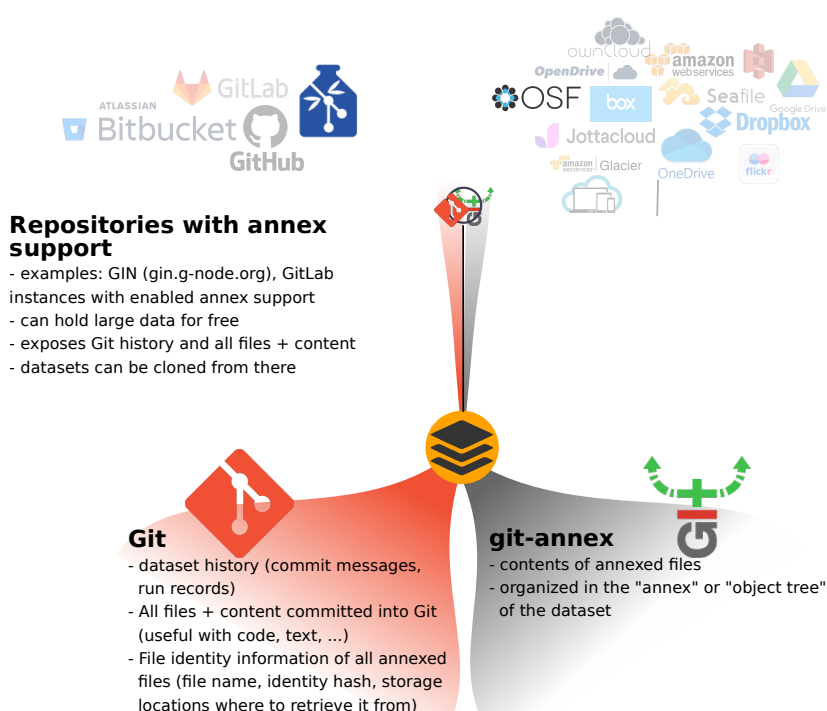
Read the section *Walk-through: Dataset hosting on GIN* (page 208) for a walk-through.

The uncommon case: Special remotes with repository hosting support

Typically, storage hosting services such as cloud storage providers do not provide the ability to host Git repositories. Therefore, it is typically not possible to **clone** from a cloud storage. However, a number of DATALAD EXTENSIONs have been created that equip cloud storage providers with the ability to also host Git repositories. While they do not get the ability to display repositories the same way that pure Git repository hosting services like GitHub do, they do get the super power of becoming clonable.

---

[199] https://git-annex.branchable.com/tips/public_Amazon_S3_remote/
[200] https://git-annex.branchable.com/tips/using_Google_Cloud_Storage/
[201] https://git-annex.branchable.com/tips/using_box.com_as_a_special_remote/
[202] https://git-annex.branchable.com/tips/using_Amazon_Glacier/
[203] https://git-annex.branchable.com/tips/owncloudannex/
[204] https://git-annex.branchable.com/special_remotes

One example for this is the Open Science Framework, which can become the home of datasets by using the datalad-osf extension[205]. As long as you and your collaborators have the extension installed, annexed dataset contents and the Git repository part of your dataset can be pushed or cloned in one go.

Publishing both components to a special remote
- performed with DataLad extensions (e.g., datalad-osf), or helpers (e.g., git-remote-rclone)
- publishes the Git repository (in two files) in conjunction with annexed data
- allows cloning from a special remote

Git
- dataset history (commit messages, run records)
- All files + content committed into Git (useful with code, text, ...)
- File identity information of all annexed files (file name, identity hash, storage locations where to retrieve it from)

git-annex
- contents of annexed files
- organized in the "annex" or "object tree" of the dataset

Please take a look at the documentation and tutorials of datalad-osf extension[206] for examples and more information.

The creative case: Ensuring availability using only repository hosting

When you only want to use pure Git repository hosting services without annex support, you can still allow others to obtain (some) file contents with some creativity:

For one, you can use commands such as **datalad download-url** (datalad-download manual) or **datalad addurls** (datalad-addurls manual) to retrieve files from web sources and register their location automatically. The first Chapter *DataLad datasets* (page 34) demonstrates **download-url**, and the usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433) demonstrates addurls on a large scale.

Other than this, you can rely on digital provenance in the form of RUN RECORDs that allow consumers of your dataset to recompute a result instead of **datalad get**ing it. The midterm-project example in section *YODA-compliant data analysis projects* (page 143) has been an example for this.

The static case: Exporting dataset snapshots

While DataLad datasets have the great advantage that they carry a history with all kinds of useful digital provenance and previous versions of files, it may not in all cases be necessary to

---

[205] http://docs.datalad.org/projects/osf/en/latest/index.html
[206] http://docs.datalad.org/projects/osf/en/latest/index.html

make use of this advantage. Sometimes, you may just want to share or archive the most recent state of the dataset as a snapshot.

DataLad provides the ability to do this out of the box to arbitrary locations, and support for specific services such as Figshare[207]. Find out more information on this in the section *Built-in data export* (page 215). Other than that, some DATALAD EXTENSIONs allow an export to additional services such as the Open Science Framework[208].

General information on publishing datasets

Beyond concrete examples of publishing datasets, some general information may be useful in addition: The section *Overview: The datalad push command* (page 219) illustrates the DataLad command **datalad push**, a command that handles every publication operation, regardless of the type of published content or its destination. In addition to this, the section *Keeping (some) dataset contents private* (page 217) contains tips and strategies on publishing datasets without leaking potentially private contents or information. Finally, you may be interested in publishing datasets into centrally managed locations for backup, archival, or central data management. In this case, take a look at the advanced section *Remote Indexed Archives for dataset storage and backup* (page 294).

13.2 Publishing datasets to Git repository hosting

Because DataLad datasets are GIT repositories, it is possible to **push** datasets to any Git repository hosting service, such as GITHUB, GITLAB, GIN, BITBUCKET, Gogs[215], or Gitea[216]. These published datasets are ordinary SIBLINGs of your dataset, and among other advantages, they can constitute a back-up, an entry-point to retrieve your dataset for others or yourself, the backbone for collaboration on datasets, or the means to enhance visibility, findability and citeability of your work[223]. This section contains a brief overview on how to publish your dataset to different services.

Git repository hosting and annexed data

As outlined in a number of sections before, Git repository hosting sites typically do not support dataset annexes - some, like GIN however, do. Depending on whether or not an annex is supported, you can push either only your Git history to the sibling, or the complete dataset including annexed file contents. You can find out whether a sibling on a remote hosting services carries an annex or not by running the **datalad siblings** command. A +, -, or ? sign in parenthesis indicates whether the sibling carries an annex, does not carry an annex, or whether this information isn't yet known. In the example below you can see that a public GitHub repository https://github.com/psychoinformatics-de/studyforrest-data-phase2 does not carry an annex on github (the sibling origin), but that the annexed data are served from an additional sibling mddatasrc (a SPECIAL REMOTE with annex support). Even though the dataset sibling on GitHub

---

[207] https://figshare.com/
[215] https://gogs.io/
[216] https://gitea.io/en-us/
[223] Many repository hosting services have useful features to make your work citeable. For example, GIN is able to assign a DOI to your dataset, and GitHub allows CITATION.cff files. At the same time, archival services such as Zenodo[224] often integrate with published repositories, allowing you to preserve your dataset with them.
[224] https://zenodo.org/

does not serve the data, it constitutes a simple, findable access point to retrieve the dataset, and can be used to provide updates and fixes via PULL REQUESTs, issues, etc.

```
# a clone of github/psychoinformatics/studyforrest-data-phase2 has the following siblings:
$ datalad siblings
.: here(+) [git]
.: mddatasrc(+) [http://psydata.ovgu.de/studyforrest/phase2/.git (git)]
.: origin(-) [git@github.com:psychoinformatics-de/studyforrest-data-phase2.git (git)]
```

There are multiple ways to create a dataset sibling on a repository hosting site to push your dataset to.

How to add a sibling on a Git repository hosting site: The manual way

1. Create a new repository via the webinterface of the hosting service of your choice. It does not need to have the same name as your local dataset, but it helps to associate local dataset and remote siblings.
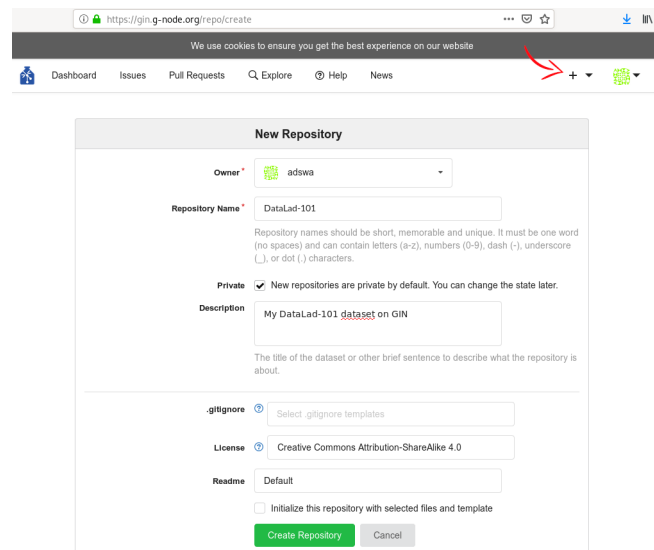
Fig. 13.3: Webinterface of GIN during the creation of a new repository.

1. Afterwards, copy the SSH or HTTPS URL of the repository. Usually, repository hosting services will provide you with a convenient way to copy it to your clipboard. An SSH URL takes the form `git@<hosting-service>:/<user>/<repo-name>.git` and an HTTPS URL takes the form `https://<hosting-service>/<user>/<repo-name>.git`. The type of URL you choose determines whether and how you will be able to push to your repository. Note that many services will require you to use the SSH URL to your repository in order to do **push** operations, so make sure to take the SSH and not the HTTPS URL if this is the case.

2. If you pick the SSH URL, make sure to have an SSH KEY set up. This usually requires generating an SSH key pair if you do not have one yet, and uploading the public key to the repository hosting service.

> **M13.2 What is an SSH key and how can I create one?**
>
> An SSH key is an access credential in the SSH protocol that can be used to login from one system to remote servers and services, such as from your private computer to an

Fig. 13.4: Webinterface of GITHUB during the creation of a new repository.

💡 SSH SERVER. For repository hosting services such as GIN, GITHUB, or GITLAB, it can be used to connect and authenticate without supplying your username or password for each action.

This tutorial by GitHub[217] is a detailed step-by-step instruction to generate and use SSH keys for authentication, and it also shows you how to add your public SSH key to your GitHub account so that you can install or clone datasets or Git repositories via SSH (in addition to the `http` protocol), and the same procedure applies to GitLab and Gin.

Don't be intimidated if you have never done this before – it is fast and easy: First, you need to create a private and a public key (an SSH key pair). All this takes is a single command in the terminal. The resulting files are text files that look like someone spilled alphabet soup in them, but constitute a secure password procedure. You keep the private key on your own machine (the system you are connecting from, and that **only you have access to**), and copy the public key to the system or service you are connecting to. On the remote system or service, you make the public key an *authorized key* to allow authentication via the SSH key pair instead of your password. This either takes a single command in the terminal, or a few clicks in a web interface to achieve. You should protect your SSH keys on your machine with a passphrase to prevent others – e.g., in case of theft – to log in to servers or services with SSH authentication[225], and configure an `ssh agent` to handle this passphrase for you with a single command. How to do all of this is detailed in the above tutorial.

---

[217] https://docs.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

[225] Your private SSH key is incredibly valuable, and it is important to keep it secret! Anyone who gets your private key has access to anything that the public key is protecting. If the private key does not have a passphrase, simply copying this file grants a person access!

1. Use the URL to add the repository as a sibling. There are two commands that allow you to do that; both require you give the sibling a name of your choice (common name choices are `upstream`, or a short-cut for your user name or the hosting platform, but its completely up to you to decide):

   1. `git remote add <name> <url>`

   2. `datalad siblings add --dataset . --name <name> --url <url>`

2. Push your dataset to the new sibling: `datalad push --to <name>`

How to add a sibling on a Git repository hosting site: The automated way

DataLad provides `create-sibling-*` commands to automatically create datasets on certain hosting sites. DataLad versions `0.16.0` and higher contain more of these commands, and provide a more streamlined parametrization. Please read the paragraph that matches your version of DataLad below, and be mindful of a change in command arguments between DataLad versions `0.15.x` and `0.16.x`.

Using DataLad version < 0.16.0

If you are using DataLad version below `0.16.0`, you can automatically create new repositories from the command line for GITHUB and GITLAB using the commands **datalad create-sibling-github** and **datalad create-sibling-gitlab**. Due to the different representation of repositories on the two sites, the two commands are parametrized differently, and it is

worth to consult each command's MANPAGE or `--help`, but below are basic usage examples for the two commands:

GitLab: Using `datalad create-sibling-gitlab` is easiest with a python-gitlab configuration. Please consult the python-gitlab documentation[218] for details, but a basic configuration in the file ~/.python-gitlab.cfg can look like this:

```
[global]
default = gitlab
ssl_verify = true
timeout = 5

[gitlab]
url = https://gitlab.myinstance.com
private_token = <super-secret-token>
api_version = 4
```

This configures the default GitLab instance (here, we have called it gitlab) with a specific base URL and the user's personal access token for authentication. Note that you will need to generate and retrieve your own personal access token under the profile settings of the gitlab instance of your choice (see the *paragraph on authentication tokens below for more information* (page 190)). With this configuration, the `--site` parameter can identify the GitLab instance by its name gitlab. If you have an SSH KEY configured, it is useful to specify `--access` as ssh – this saves you the need to authenticate with every push:

```
$ datalad create-sibling-gitlab \
  -d . \                      # current dataset
  --site gitlab \             # to the configured GitLab instance
  --project DataLad-101 \     # repository name
  --layout flat \
  --access ssh                # optional, but useful
    create_sibling_gitlab(ok): . (dataset)
    configure-sibling(ok): . (sibling)
    action summary:
      configure-sibling (ok: 1)
      create_sibling_gitlab (ok: 1)
$ datalad siblings
  here(+) [git]
      jugit(-) [git@gitlab.myinstance.com:<user>/<repo>.git (git)]
$ datalad push --to gitlab
    publish(ok): . (dataset)
    action summary:
      publish (ok: 1)
```

GitHub: The command `datalad create-sibling-github` requires a personal access token from GitHub (see the *paragraph on authentication tokens below for more information* (page 190)). When you are using it for the first time, you should be queried interactively for it. Subsequently, your token should be stored internally.

By default, the URL that is set up for you is an HTTPS URL. If you have an SSH KEY configured, it is useful to specify `--access-protocol` as ssh – with this the SSH URL is configured, saving you the need to authenticate with every push.

---

[218] https://python-gitlab.readthedocs.io/en/stable/cli-usage.html#configuration

```
$ datalad create-sibling-github \
  -d . \                         # current dataset
  DataLad-101 \                  # repository name
  --access-protocol ssh          # optional, but useful
You need to authenticate with 'github' credentials. https://github.com/settings/tokens␣
↪provides information on how to gain access
token: <my-super-secret-token>
create_sibling_github(ok): . (dataset) [Dataset sibling 'github', project at https://
↪github.com/adswa/DataLad-101.git]
configure-sibling(ok): . (sibling)
action summary:
  configure-sibling (ok: 1)
  create_sibling_github (ok: 1)
$ datalad push --to github
publish(ok): . (dataset)
action summary:
  publish (ok: 1)
```

Using DataLad version 0.16.0 and higher

Starting with DataLad version `0.16.0` or higher, you can automatically create new repositories from the command line for GITHUB, GITLAB, GIN, Gogs[219], or Gitea[220]. This is implemented with a new set of commands called **create-sibling-github**, **create-sibling-gitlab**, **create-sibling-gin**, **create-sibling-gogs**, and **create-sibling-gitea**.

> **G13.1 Get DataLad features ahead of time by installing from a commit**
>
> If you want to get this feature ahead of the `0.16.0` release, you can install the most recent version of the MASTER BRANCH or a specific COMMIT hash from GitHub, for example with
>
> ```
> $ pip install git+git://github.com/datalad/datalad.git@master
> ```
>
> When getting features ahead of time, your feedback is especially valuable. If you find that something does not work, or if you have an idea for improvements, please get in touch[221].
>
> ---
> [221] https://github.com/datalad/datalad/issues/new

Each command is slightly tuned towards the peculiarities of each particular platform, but the most important common parameters are streamlined across commands as follows:

- `[REPONAME]` (required): The name of the repository on the hosting site. It will be created under a user's namespace, unless this argument includes an organization name prefix. For example, `datalad create-sibling-github my-awesome-repo` will create a new repository under `github.com/<user>/my-awesome-repo`, while `datalad create-sibling-github <orgname>/my-awesome-repo` will create a new repository of this name under the GitHub organization `<orgname>` (given appropriate permissions).

- `-s/--name <name>` (required): A name under which the sibling is identified. By default, it will be based on or similar to the hosting site. For example, the sibling created with `datalad create-sibling-github` will be called `github` by default.

---
[219] https://gogs.io/
[220] https://gitea.io/en-us/

- `--credential <name>` (optional): Credentials used for authentication are stored internally by DataLad under specific names. These names allow you to have multiple credentials, and flexibly decide which one to use. When `--credential <name>` is the name of an existing credential, DataLad tries to authenticate with the specified credential; when it does not yet exist DataLad will prompt interactively for a credential, such as an access token, and store it under the given `<name>` for future authentications. By default, DataLad will name a credential according to the hosting service URL it used for, for example `datalad-api.github.com` as the default for credentials used to authenticate against GitHub.

- `--access-protocol {https|ssh|https-ssh}` (default `https`): Whether to use SSH or HTTPS URLs, or a hybrid version in which HTTPS is used to *pull* and SSH is used to *push*. Using SSH URLs requires an SSH KEY setup, but is a very convenient authentication method, especially when pushing updates – which would need manual input on user name and token with every push over HTTPS.

- `--dry-run` (optional): With this flag set, the command will not actually create the target repository, but only perform tests for name collisions and report repository name(s).

- `--private` (optional): A switch that, if set, makes sure that the created repository is private.

Other streamlined arguments, such as `--recursive` or `--publish-depends` allow you to perform more complex configurations, for example publication of dataset hierarchies or connections to SPECIAL REMOTEs. Upcoming walk-throughs will demonstrate them.

Self-hosted repository services, e.g., Gogs or Gitea instances, have an additional required argument, the `--api` flag. It needs to point to the URL of the instance, for example

```
$ datalad create-sibling-gogs my_repo_on_gogs  --api "https://try.gogs.io"
```

Authentication by token

To create or update repositories on remote hosting services you will need to set up appropriate authentication and permissions. In most cases, this will be in the form of an authorization token with a specific permission scope.

What is a token?

Personal access tokens are an alternative to authenticating via your password, and take the form of a long character string, associated with a human-readable name or description. If you are prompted for `username` and `password` in the command line, you would enter your token in place of the `password`[226]. Note that you do not have to type your token at every authentication – your token will be stored on your system the first time you have used it and automatically reused whenever relevant.

M13.3 How does the authentication storage work?

Passwords, user names, tokens, or any other login information is stored in your system's (encrypted) keyring[222]. It is a built-in credential store, used in all major operating sys-

---

[226] GitHub deprecated user-password authentication[227] and only supports authentication via personal access token from November 13th 2020 onwards. Supplying a password instead of a token will fail to authenticate.

[227] https://developer.github.com/changes/2020-02-14-deprecating-password-auth/

> 💡 tems, and can store credentials securely.
> _____
> [222] https://en.wikipedia.org/wiki/GNOME_Keyring

You can have multiple tokens, and each of them can get a different scope of permissions, but it is important to treat your tokens like passwords and keep them secret.

Which permissions do they need?

The most convenient way to generate tokens is typically via the webinterface of the hosting service of your choice. Often, you can specifically select which set of permissions a specific token has in a drop-down menu similar (but likely not identical) to this screenshot from GitHub:

For creating and updating repositories with DataLad commands it is usually sufficient to grant only repository-related permissions. However, broader permission sets may also make sense. Should you employ GitHub workflows, for example, a token without "workflow" scope could not push changes to workflow files, resulting in errors like this one:

```
[remote rejected] (refusing to allow a Personal Access Token to create or update workflow
→`.github/workflows/benchmarks.yml` without `workflow` scope)]
```

13.3 Walk-through: Dropbox as a special remote

Let's say you'd like to share your complete `DataLad-101` dataset with a friend overseas. After all you know about DataLad, you'd like to let more people know about its capabilities. You and your friend, however, do not have access to the same computational infrastructure, and there are also many annexed files, e.g., the PDFs in your dataset, that you'd like your friend to have but that can't be simply computed or automatically obtained from web sources. What you would like to do is to provide your friend with a URL to install a dataset from *and* successfully run **datalad get**, just as with the many publicly available DataLad datasets such as the `longnow` podcasts.

As an example, let's walk through all necessary steps to publish the `DataLad-101` dataset to GitHub, and its file contents to **Dropbox**. To make this as convenient as possible, we will also set up a PUBLICATION DEPENDENCY between the two.

To set up Dropbox as a third party storage provide you need to configure a special-remote called rclone[228]. It is a command line program to sync files and directories to and from a large number of commercial providers[237].

- The first step is to install[229] `rclone` on your computer. The installation instructions are straightforward and the installation is quick if you are on a Unix-based system (macOS or any Linux distribution).

- Afterwards, run `rclone config` from the command line to configure `rclone` to work with Dropbox. Running this command will a guide you with an interactive prompt through

_____
[228] https://github.com/DanielDent/git-annex-remote-rclone
[237] `rclone` is a useful special-remote for this example, because you can not only use it for Dropbox, but also for many other third-party hosting services. For a complete overview of which third-party services are available and which special-remote they need, please see this list[238].
[238] http://git-annex.branchable.com/special_remotes/
[229] https://rclone.org/install/

Fig. 13.5: Webinterface to generate an authentication token on GitHub. One typically has to set a name and permission set, and potentially an expiration date.

a ~2 minute configuration of the remote (here we will name the remote "dropbox-for-friends" – the name will be used to refer to it later during the configuration of the dataset we want to publish). The interactive dialog is outlined below, and all parts that require user input are highlighted.

```
$ rclone config
 2019/09/06 13:43:58 NOTICE: Config file "/home/me/.config/rclone/rclone.conf" not found -
↪ using defaults
 No remotes found - make a new one
 n) New remote
 s) Set configuration password
 q) Quit config
 n/s/q> n
 name> dropbox-for-friends
 Type of storage to configure.
 Enter a string value. Press Enter for the default ("").
 Choose a number from below, or type in your own value
  1 / 1Fichier
    \ "fichier"
  2 / Alias for an existing remote
    \ "alias"
 [...]
  8 / Dropbox
    \ "dropbox"
 [...]
 31 / premiumize.me
    \ "premiumizeme"
 Storage> dropbox
 ** See help for dropbox backend at: https://rclone.org/dropbox/ **

 Dropbox App Client Id
 Leave blank normally.
 Enter a string value. Press Enter for the default ("").
 client_id>
 Dropbox App Client Secret
 Leave blank normally.
 Enter a string value. Press Enter for the default ("").
 client_secret>
 Edit advanced config? (y/n)
 y) Yes
 n) No
 y/n> n
 If your browser doesn't open automatically go to the following link: http://127.0.0.
↪1:53682/auth
 Log in and authorize rclone for access
 Waiting for code...
```

- At this point, this will open a browser and ask you to authorize `rclone` to manage your Dropbox, or any other third-party service you have selected in the interactive prompt. Accepting will bring you back into the terminal to the final configuration prompts:

```
Got code
--------------------
[dropbox-for-friends]
type = dropbox
token = {"access_token":"meVHyc[...]",
        "token_type":"bearer",
```

<div align="right">(continues on next page)</div>

```
        "expiry":"0001-01-01T00:00:00Z"}
-------------------
y) Yes this is OK
e) Edit this remote
d) Delete this remote
y/e/d> y
Current remotes:

Name                    Type
====                    ====
dropbox-for-friends     dropbox

e) Edit existing remote
n) New remote
d) Delete remote
r) Rename remote
c) Copy remote
s) Set configuration password
q) Quit config
e/n/d/r/c/s/q> q
```

- Once this is done, install `git-annex-remote-rclone`. It is a wrapper around rclone[230] that makes any destination supported by rclone usable with Git-Annex. If you are on a recent version of Debian or Ubuntu (or have enabled the NeuroDebian[231] repository), you can get it conveniently via your package manager, e.g., with `sudo apt-get install git-annex-remote-rclone`. Alternatively, `git clone` the git-annex-remote-rclone[232] repository to your machine (do not clone it into `DataLad-101` but somewhere else on your computer), and copy the path to this repository into your `$PATH` variable. If you clone into `/home/user-bob/repos`, the command would look like this[239]:

```
$ git clone https://github.com/DanielDent/git-annex-remote-rclone.git
$ export PATH="/home/user-bob/repos/git-annex-remote-rclone:$PATH"
```

- Finally, in the dataset you want to share, run the **`git annex initremote`** command. Give the remote a name (it is `dropbox-for-friends` here), and specify the name of the remote you configured with `rclone` with the `target` parameters:

```
$ git annex initremote dropbox-for-friends type=external externaltype=rclone chunk=50MiB␣
↪encryption=none target=dropbox-for-friends prefix=my_awesome_dataset

initremote dropbox-for-friends ok
(recording state in git...)
```

What has happened up to this point is that we have configured Dropbox as a third-party storage service for the annexed contents in the dataset. On a conceptual, dataset level, your Dropbox folder is now a SIBLING – the sibling name is the first positional argument after `initremote`, i.e., "dropbox-for-friends":

---

[230] https://rclone.org

[231] https://neuro.debian.net

[232] https://github.com/DanielDent/git-annex-remote-rclone

[239] Note that `export` will extend your `$PATH` *for your current shell*. This means you will have to repeat this command if you open a new shell. Alternatively, you can insert this line into your shells configuration file (e.g., `~/.bashrc`) to make this path available to all future shells of your user account. If you are unsure what any of this means, take a look at *this additional information on environment variables* (page 124)

```
$ datalad siblings
 .: here(+) [git]
 .: dropbox-for-friends(+) [rclone]
 .: roommate(+) [../mock_user/DataLad-101 (git)]
```

On Dropbox, a new folder will be created for your annexed files. By default, this folder will be
called git-annex, but it can be configured using the --prefix=<whatitshouldbecalled> option,
as done above. However, this directory on Dropbox is not the location you would refer your
friend or a collaborator to. The representation of the files in the special-remote is not human-
readable – it is a tree of annex objects, and thus looks like a bunch of very weirdly named
folders and files to anyone. Through this design it becomes possible to chunk files into smaller
units (see the git-annex documentation[233] for more on this), optionally encrypt content on its
way from a local machine to a storage service (see the git-annex documentation[234] for more on
this), and avoid leakage of information via file names. Therefore, the Dropbox remote is not
a places a real person would take a look at, instead they are only meant to be managed and
accessed via DataLad/git-annex.

To actually share your dataset with someone, you need to *publish* it to Github, Gitlab, or a
similar hosting service.

You could, for example, create a sibling of the DataLad-101 dataset on GitHub with the com-
mand **create-sibling-github**. This will create a new GitHub repository called "DataLad-101"
under your account, and configure this repository as a SIBLING of your dataset called github
(exactly like you have done in *YODA-compliant data analysis projects* (page 143) with the
midterm_project subdataset). However, in order to be able to link the contents stored in Drop-
box, you also need to configure a *publication dependency* to the dropbox-for-friends sibling –
this is done with the publish-depends <sibling> option.

```
$ datalad create-sibling-github -d . DataLad-101 \
  --publish-depends dropbox-for-friends
 [INFO   ] Configure additional publication dependency on "dropbox-for-friends"
 .: github(-) [https://github.com/<user-name>/DataLad-101.git (git)]
 'https://github.com/<user-name>/DataLad-101.git' configured as sibling 'github' for
↪<Dataset path=/home/me/dl-101/DataLad-101>
```

datalad siblings will again list all available siblings:

```
$ datalad siblings
 .: here(+) [git]
 .: dropbox-for-friends(+) [rclone]
 .: roommate(+) [../mock_user/DataLad-101 (git)]
 .: github(-) [https://github.com/<user-name>/DataLad-101.git (git)]
```

Note that each sibling has either a + or - attached to its name. This indicates the presence (+) or
absence (-) of a remote data annex at this remote. You can see that your github sibling indeed
does not have a remote data annex. Therefore, instead of "only" publishing to this GitHub repos-
itory (as done in section *YODA-compliant data analysis projects* (page 143)), in order to also
publish annex contents, we made publishing to GitHub dependent on the dropbox-for-friends
sibling (that has a remote data annex), so that annexed contents are published there first.

---

[233] https://git-annex.branchable.com/chunking/
[234] https://git-annex.branchable.com/encryption/

> ⚠️ **Publication dependencies are strictly local configuration**
>
> Note that the publication dependency is only established for your own dataset, it is not shared with clones of the dataset. Internally, this configuration is a key value pair in the section of your remote in `.git/config`:
>
> ```
> [remote "github"]
>     annex-ignore = true
>     url = https://github.com/<user-name>/DataLad-101.git
>     fetch = +refs/heads/*:refs/remotes/github/*
>     datalad-publish-depends = dropbox-for-friends
> ```

With this setup, we can publish the dataset to GitHub. Note how the publication dependency is served first:

```
$ datalad push --to github
[INFO   ] Transferring data to configured publication dependency: 'dropbox-for-friends'
[INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101> data to dropbox-for-
↪friends
publish(ok): books/TLCL.pdf (file)
publish(ok): books/byte-of-python.pdf (file)
publish(ok): books/progit.pdf (file)
publish(ok): recordings/interval_logo_small.jpg (file)
publish(ok): recordings/salt_logo_small.jpg (file)
[INFO   ] Publishing to configured dependency: 'dropbox-for-friends'
[INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101> data to dropbox-for-
↪friends
[INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101> to github
Username for 'https://github.com': <user-name>
Password for 'https://<user-name>@github.com':
publish(ok): . (dataset) [pushed to github: ['[new branch]', '[new branch]']]
action summary:
  publish (ok: 6)
```

Afterwards, your dataset can be found on GitHub, and `cloned` or `installed`.

From the perspective of whom you share your dataset with…

If your friend would now want to get your dataset including the annexed contents, and you made sure that they can access the Dropbox folder with the annexed files (e.g., by sharing an access link), here is what they would have to do:

If the repository is on GitHub, a **datalad clone** with the URL will install the dataset:

```
$ datalad clone https://github.com/<user-name>/DataLad-101.git
[INFO   ] Cloning https://github.com/<user-name>/DataLad-101.git [1 other candidates]␣
↪into '/Users/awagner/Documents/DataLad-101'
[INFO   ]   Remote origin not usable by git-annex; setting annex-ignore
[INFO   ] access to 1 dataset sibling dropbox-for-friends not auto-enabled, enable with:
|         datalad siblings -d "/Users/awagner/Documents/DataLad-101" enable -s dropbox-
↪for-friends
install(ok): /Users/awagner/Documents/DataLad-101 (dataset)
```

Pay attention to one crucial information in this output:

```
[INFO   ] access to 1 dataset sibling dropbox-for-friends not auto-enabled, enable with:
|        datalad siblings -d "/Users/<user-name>/Documents/DataLad-101" enable -s␣
↪dropbox-for-friends
```

This means that someone who wants to access the data from dropbox needs to enable the special remote. For this, this person first needs to install and configure `rclone` as well: Since `rclone` is the protocol with which annexed data can be transferred from and to Dropbox, anyone who needs annexed data from Dropbox needs *this* special remote. Therefore, the first steps are the same as before:

- Install[235] `rclone` (as described above).

- Run `rclone config` to configure `rclone` to work with Dropbox (as described above). **It is important to name the remote identically** - in the example above, it would need to be "dropbox-for-friends". This means: You need to communicate the name of your special remote to your friend, and they have to give it the same name as the one configured in the dataset). (There are efforts towards extracting this information automatically from datasets, but for the time being, this is an important detail to keep in mind).

- install git-annex-remote-rclone[236] (as described above).

After this is done, you can execute what DataLad's output message suggests to "enable" this special remote (inside of the installed `DataLad-101`):

```
$ datalad siblings -d "/Users/awagner/Documents/DataLad-101" \
  enable -s dropbox-for-friends
.: dropbox-for-friends(?) [git]
```

And once this is done, you can get any annexed file contents, for example the books, or the cropped logos from chapter *DataLad, Run!* (page 59):

```
$ datalad get books/TLCL.pdf
get(ok): /home/some/other/user/DataLad-101/books/TLCL.pdf (file) [from dropbox-for-
↪friends]
```

13.4 Walk-through: Amazon S3 as a special remote

Amazon S3[240] (or Amazon Simple Storage Service) is a popular service by Amazon Web Services[241] (AWS) that provides object storage through a web service interface. An S3 bucket can be configured as a GIT-ANNEX SPECIAL REMOTE, allowing it to be used as a DataLad publication target. This means that you can use Amazon S3 to store your annexed data contents and allow users to install your full dataset with DataLad from a publicly available repository service such as GitHub.

> **M13.4 What is a special remote**
>
> A special-remote is an extension to Git's concept of remotes, and can enable GIT-ANNEX to transfer data from and possibly to places that are not Git repositories (e.g., cloud services or external machines such as an HPC system). For example, *s3* special remote uploads

---

[235] https://rclone.org/install/
[236] https://github.com/DanielDent/git-annex-remote-rclone
[240] https://aws.amazon.com/s3/
[241] https://aws.amazon.com/

> 💡 and downloads content to AWS S3, *web* special remote downloads files from the web, *datalad-archive* extracts files from the annexed archives, etc. Don't envision a special-remote as merely a physical place or location – a special-remote is a protocol that defines the underlying transport of your files to and/or from a specific location.

In this section, we provide a walkthrough on how to set up Amazon S3 for hosting your DataLad dataset, and how to access this data locally from GitHub.

Prerequisites

In order to use Amazon S3 for hosting your datasets, and to follow the steps below, you need to:

- Signup for an AWS account[242]
- Verify your account
- Find your AWS access key
- Signup for a GitHub account[243]
- Install wget[244] in order to download sample data
- Optional: install the AWS Command Line Interface[245]

The AWS signup[246] procedure requires you to provide your e-mail address, physical address, and credit card details before verification is possible.

> ⚠️ **AWS account usage can incur costs**
>
> While Amazon provides Free Tier[247] access to its services, it can still potentially result in costs if usage exceeds Free Tier Limits[248]. Be sure to take note of these limits, or set up automatic tracking alerts[249] to be notified before incurring unnecessary costs.
>
> ---
> [247] https://aws.amazon.com/free/
> [248] https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/free-tier-limits.html
> [249] https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/tracking-free-tier-usage.html

To find your AWS access key, log in to the AWS Console[250], open the dropdown menu at your username (top right), and select "My Security Credentials". A new page will open with several options, including "Access keys (access key ID and secret access key)" from where you can select "Create New Access Key" or access existing credentials. Take note to copy both the "Access Key ID" and "Secret Access Key".

To ensure that your access key details are known when initializing the special remote, export them as ENVIRONMENT VARIABLEs in your shell:

```
$ export AWS_ACCESS_KEY_ID="<your-access-key-ID>"
$ export AWS_SECRET_ACCESS_KEY"<your-secret-access-key>"
```

---
[242] https://aws.amazon.com/
[243] https://github.com/join
[244] https://www.gnu.org/software/wget/
[245] https://aws.amazon.com/cli/
[246] https://aws.amazon.com/
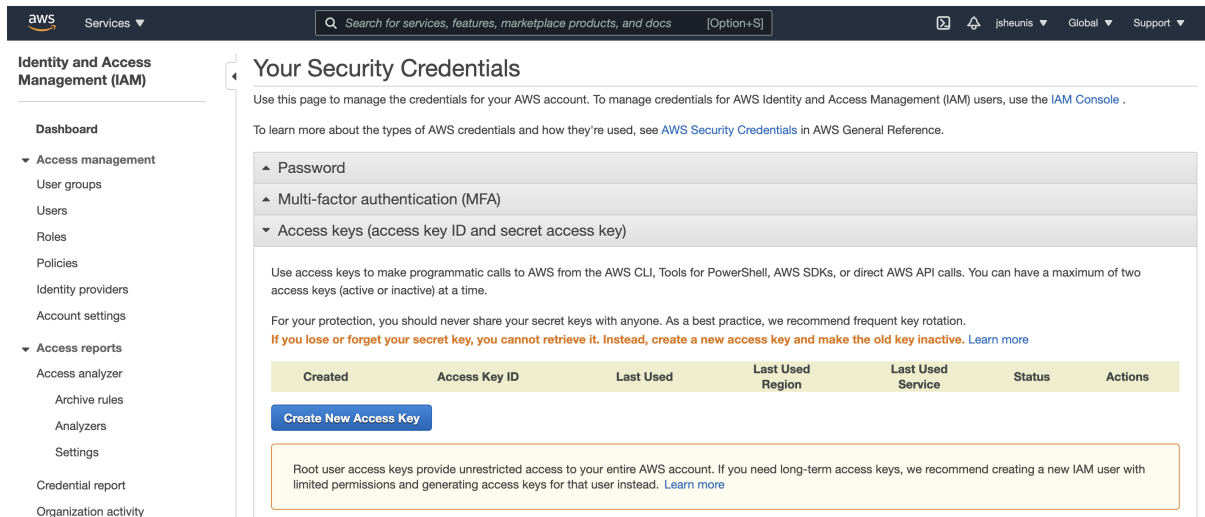[250] https://console.aws.amazon.com/

Fig. 13.6: Create a new AWS access key from "My Security Credentials"

In order to work directly with AWS via your command-line shell, you can install the AWS CLI[251].
However, that is not required for this walkthrough.

Lastly, to publish your data repository to GitHub, from which users will be able do install the
complete dataset, you will need a GitHub account[252].

Your DataLad dataset

If you already have a small DataLad dataset to practice with, feel free to use it during the rest
of the walkthrough. If you do not have data, no problem! As a general introduction, the steps
below will download a small public neuroimaging dataset, and transform it into a DataLad
dataset. We'll use the MoAEpilot[253] dataset containing anatomical and functional images from
a single subject, as well as some metadata.

In the first step, we create a new directory called `neuro-data-s3`, we download and extract the
data, and then we move the extracted contents into our new directory:

```
$ cd <wherever-you-want-to-create-the-dataset>
$ mkdir neuro-data-s3 && \
wget https://www.fil.ion.ucl.ac.uk/spm/download/data/MoAEpilot/MoAEpilot.bids.zip -O␣
↪neuro-data-s3.zip && \
unzip neuro-data-s3.zip -d neuro-data-s3 && \
rm neuro-data-s3.zip && \
cd neuro-data-s3 && \
mv MoAEpilot/* . && \
rm -R MoAEpilot

--2021-06-01 09:32:25--  https://www.fil.ion.ucl.ac.uk/spm/download/data/MoAEpilot/
↪MoAEpilot.bids.zip
Resolving www.fil.ion.ucl.ac.uk (www.fil.ion.ucl.ac.uk)... 193.62.66.18
Connecting to www.fil.ion.ucl.ac.uk (www.fil.ion.ucl.ac.uk)|193.62.66.18|:443...␣
↪connected.
```

(continues on next page)

---

[251] https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html
[252] https://github.com/join
[253] https://www.fil.ion.ucl.ac.uk/spm/data/auditory/

```
HTTP request sent, awaiting response... 200 OK
Length: 30176409 (29M) [application/zip]
Saving to: 'neuro-data-s3.zip'

neuro-data-s3.zip                                               100
↪%[=====================================================================================
↪]  28.78M  55.3MB/s    in 0.5s

2021-06-01 09:32:25 (55.3 MB/s) - 'neuro-data-s3.zip' saved [30176409/30176409]

Archive:  neuro-data-s3.zip
   creating: neuro-data-s3/MoAEpilot/
inflating: neuro-data-s3/MoAEpilot/task-auditory_bold.json
inflating: neuro-data-s3/MoAEpilot/README
inflating: neuro-data-s3/MoAEpilot/dataset_description.json
inflating: neuro-data-s3/MoAEpilot/CHANGES
   creating: neuro-data-s3/MoAEpilot/sub-01/
   creating: neuro-data-s3/MoAEpilot/sub-01/func/
inflating: neuro-data-s3/MoAEpilot/sub-01/func/sub-01_task-auditory_events.tsv
inflating: neuro-data-s3/MoAEpilot/sub-01/func/sub-01_task-auditory_bold.nii
   creating: neuro-data-s3/MoAEpilot/sub-01/anat/
inflating: neuro-data-s3/MoAEpilot/sub-01/anat/sub-01_T1w.nii
```

Now we can view the directory tree to see the dataset content:

```
$ tree
.
├── CHANGES
├── README
├── dataset_description.json
├── sub-01
│   ├── anat
│   │   └── sub-01_T1w.nii
│   └── func
│       ├── sub-01_task-auditory_bold.nii
│       └── sub-01_task-auditory_events.tsv
└── task-auditory_bold.json
```

The next step is to ensure that this is a valid DataLad dataset, with `main` as the default branch.

> ⚠️ **Ensure main is set as default branch for newly-created repositories**
>
> Any new dataset configured with `master` instead of `main` as the default branch will get `git-annex` configured to be the default displayed branch when it is pushed to GitHub. See *this FAQ for more information* (page 487). This can be prevented by:
> - a user/organization setting on GitHub about default branches[254]
> - setting `main` as the default branch by changing your global `git config`:
>
> ```
> git config --global init.defaultBranch main
> ```
>
> ---
> [254] https://github.blog/changelog/2020-08-26-set-the-default-branch-for-newly-created-repositories/

We can turn our `neuro-data-s3` directory into a DataLad dataset with the **datalad create --force** command. After that, we save the dataset with **datalad save**:

```
$ datalad create --force --description "neuro data to host on s3"
[INFO   ] Creating a new annex repo at /Users/jsheunis/Documents/neuro-data-s3
[INFO   ] Scanning for unlocked files (this may take some time)
create(ok): /Users/jsheunis/Documents/neuro-data-s3 (dataset)

$ datalad save -m "Add public data"
add(ok): CHANGES (file)
add(ok): README (file)
add(ok): dataset_description.json (file)
add(ok): sub-01/anat/sub-01_T1w.nii (file)
add(ok): sub-01/func/sub-01_task-auditory_bold.nii (file)
add(ok): sub-01/func/sub-01_task-auditory_events.tsv (file)
add(ok): task-auditory_bold.json (file)
save(ok): . (dataset)
action summary:
  add (ok: 7)
  save (ok: 1)
```

Initialize the S3 special remote

The steps below have been adapted from instructions provided on git-annex documentation[255].

By initializing the special remote, what actually happens in the background is that a SIBLING is added to the DataLad dataset. This can be verified by running **datalad siblings** before and after initializing the special remote. Before, the only "sibling" is the actual DataLad dataset:

```
$ datalad siblings
.: here(+) [git]
```

To initialize a public S3 bucket as a special remote, we run **git annex initremote** with several options, for which git-annex documentation on S3[256] provides detailed information. Be sure to select a unique bucket name that adheres to Amazon S3's bucket naming rules[257]. You can declare the bucket name (in this example "sample-neurodata-public") as a variable since it will be used again later.

```
$ BUCKET=sample-neurodata-public
$ git annex initremote public-s3 type=S3 encryption=none \
bucket=$BUCKET public=yes datacenter=EU autoenable=true
initremote public-s3 (checking bucket...) (creating bucket in EU...) ok
(recording state in git...)
```

The options used in this example include:

- public-s3: the name we select for our special remote, so that git-annex and DataLad can identify it

- type=S3: the type of special remote (git-annex can work with many special remote types[258])

- encryption=none: no encryption (alternatively enable encryption=shared, meaning files will be encrypted on S3, and anyone with a clone of the git repository will be able to

---

[255] https://git-annex.branchable.com/tips/public_Amazon_S3_remote/
[256] https://git-annex.branchable.com/special_remotes/S3/
[257] https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucketnamingrules.html
[258] https://git-annex.branchable.com/special_remotes/

download and decrypt them)

- `bucket=$BUCKET`: the name of the bucket to be created on S3 (using the declared variable)

- `public=yes`: Set to "yes" to allow public read access to files sent to the S3 remote

- `datacenter=EU`: specify where the data will be located; here we set "EU" which is EU/Ireland a.k.a. `eu-west-1` (defaults to "US" if not specified)

- `autoenable=true`: git-annex will attempt to enable the special remote when it is run in a new clone, implying that users won't have to run extra steps when installing the dataset with DataLad

After `git annex initremote` has successfully initialized the special remote, you can run `datalad siblings` to see that a sibling has been added:

```
$ datalad siblings
.: here(+) [git]
.: public-s3(+) [git]
```

You can also visit the S3 Console[259] and navigate to "Buckets" to see your newly created bucket. It should only have a single `annex-uuid` file as content, since no actual file content has been pushed yet.
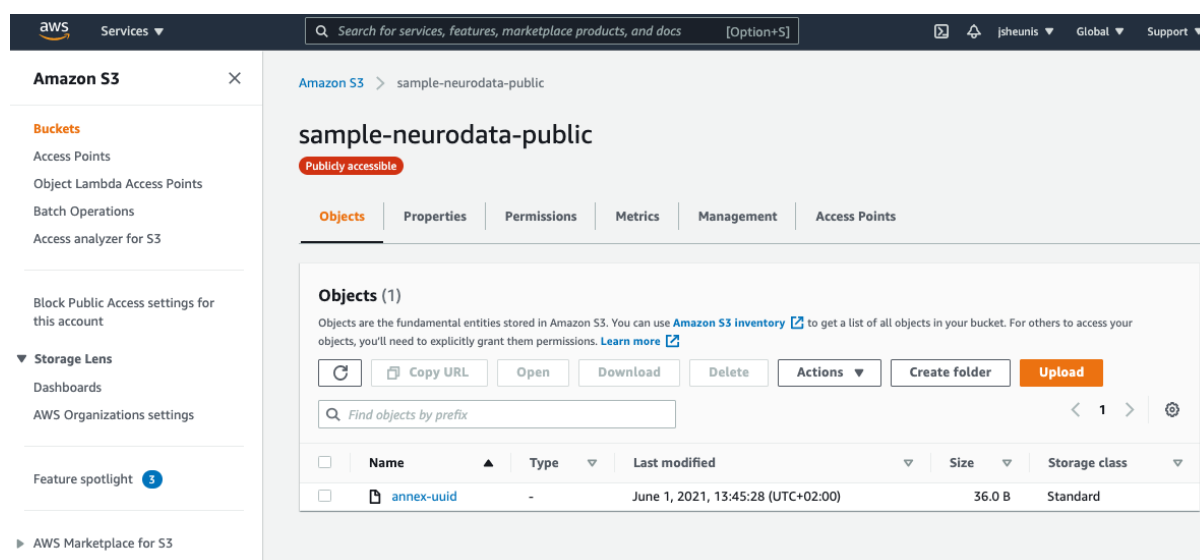
Fig. 13.7: A newly created public S3 bucket

Lastly, for git-annex to be able to download files from the bucket without requiring your AWS credentials, it needs to know where to find the bucket. We do this by setting the bucket URL, which takes a standard format incorporating the bucket name and location (see the code block below). Alternatively, this URL can also be copied from your AWS console.

```
$ git annex enableremote public-s3 \
publicurl="https://$BUCKET.s3-eu-west-1.amazonaws.com"
enableremote public-s3 ok
(recording state in git...)
```

---

[259] https://console.aws.amazon.com/s3/

Publish the dataset

The special remote is ready, and now we want to give people seamless access to the Data-Lad dataset. A common way to do this is to create a sibling of the dataset on GitHub using **create-sibling-github**. In order to link the contents in the S3 special remote to the GitHub sibling, we also need to configure a publication dependency to the `public-s3` sibling, which is done with the `publish-depends <sibling>` option. For consistency, we'll give the GitHub repository the same name as the dataset name.

```
$ datalad create-sibling-github -d . neuro-data-s3 \
--publish-depends public-s3
[INFO   ] Configure additional publication dependency on "public-s3"
.: github(-) [https://github.com/jsheunis/sample-neuro-data.git (git)]
'https://github.com/jsheunis/sample-neuro-data.git' configured as sibling 'github' for␣
→Dataset(/Users/jsheunis/Documents/neuro-data-s3)
```

Notice that by creating this sibling, DataLad created an actual (empty) dataset repository on GitHub, which required preconfigured GitHub authentication details.

> ⚠️ **GitHub deprecated its User Password authentication**
>
> GitHub decided to deprecate user-password authentication[260] and only supports authentication via personal access token from November 13th 2020 onwards. Changes in DataLad's API reflect this change starting with DataLad version `0.13.6` by removing the `github-passwd` argument. Starting with DataLad `0.16.0`, a new set of commands for interactions with a variety of hosting services will be introduced (for more information, see section *Publishing datasets to Git repository hosting* (page 184)).
>
> To ensure successful authentication, please create a personal access token at github.com/settings/tokens[261,264], and either
>
> - supply the token with the argument `--github-login <TOKEN>` from the command line,
> - or supply the token from the command line when queried for a password
>
> ---
>
> [260] https://developer.github.com/changes/2020-02-14-deprecating-password-auth/
> [261] https://github.com/settings/tokens
> [264] Instead of using GitHub's WebUI you could also obtain a token using the command line GitHub interface (https://github.com/sociomantic-tsunami/git-hub) by running `git hub setup` (if no 2FA is used). If you decide to use the command line interface, here is help on how to use it: Clone the GitHub repository[265] to your local computer. Decide whether you want to build a Debian package to install, or install the single-file Python script distributed in the repository. Make sure that all requirements[266] for your preferred version are installed , and run either `make deb` followed by `sudo dpkg -i deb/git-hub*all.deb`, or `make install`.

The creation of the sibling (named `github`) can also be confirmed with **datalad siblings**:

```
$ datalad siblings
.: here(+) [git]
.: public-s3(+) [git]
.: github(-) [https://github.com/jsheunis/neuro-data-s3.git (git)]
```

The next step is to actually push the file content to where it needs to be in order to allow others to access the data. We do this with **datalad push --to github**. The `--to github` specifies which sibling to push the dataset to, but because of the publication dependency DataLad will push the annexed contents to the special remote first.

```
$ datalad push --to github
copy(ok): CHANGES (file) [to public-s3...]
copy(ok): README (file) [to public-s3...]
copy(ok): dataset_description.json (file) [to public-s3...]
copy(ok): sub-01/anat/sub-01_T1w.nii (file) [to public-s3...]
copy(ok): sub-01/func/sub-01_task-auditory_bold.nii (file) [to public-s3...]
copy(ok): sub-01/func/sub-01_task-auditory_events.tsv (file) [to public-s3...]
copy(ok): task-auditory_bold.json (file) [to public-s3...]
publish(ok): . (dataset) [refs/heads/main->github:refs/heads/main [new branch]]
publish(ok): . (dataset) [refs/heads/git-annex->github:refs/heads/git-annex [new branch]]
```

You can now view the annexed file content (with MD5 hashes as filenames) in the S3 bucket[262]:
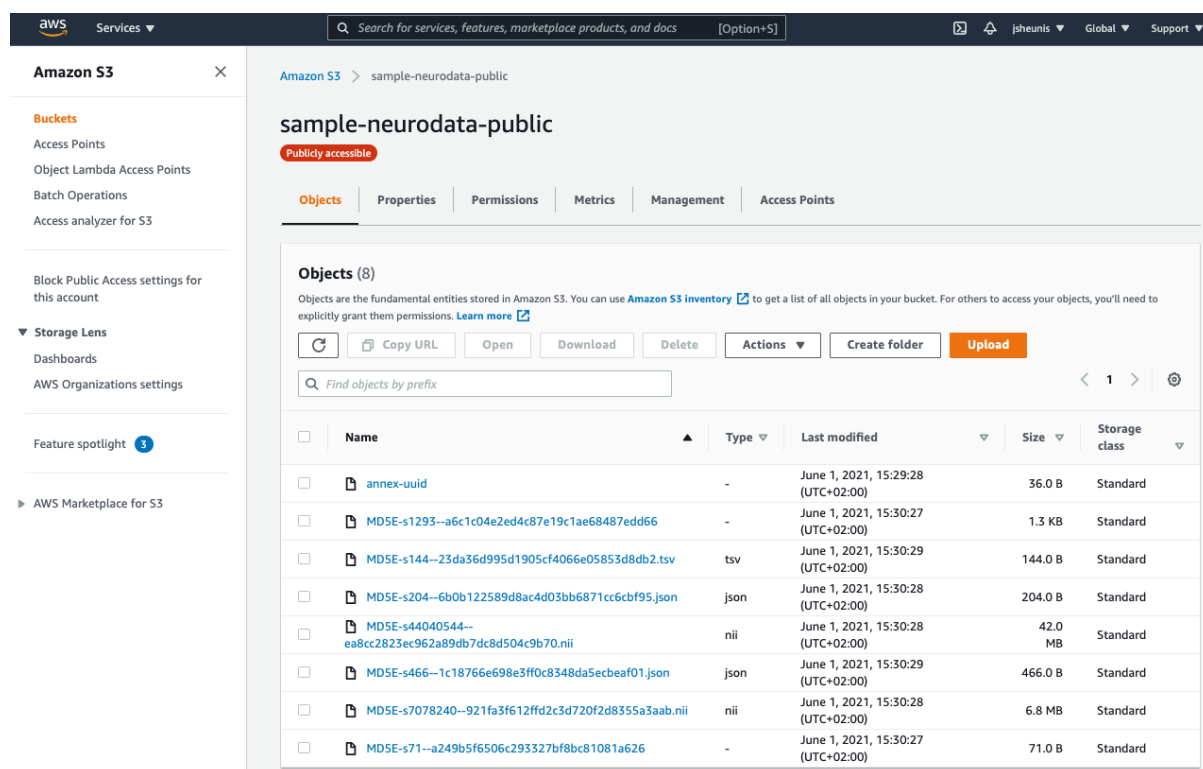
Fig. 13.8: The public S3 bucket with annexed file content pushed

Lastly, the GitHub repository will also show the newly pushed dataset (with the "files" being symbolic links to the annexed content on the S3 remote):

Test the setup!

You have now successfully created a DataLad dataset with an AWS S3 special remote for annexed file content and with a public GitHub sibling from which the dataset can be accessed. Users can now `datalad clone` the dataset using the GitHub repository URL:

```
$ cd /tmp
$ datalad clone https://github.com/<enter-your-your-organization-or-account-name-here>/
↪neuro-data-s3.git
[INFO   ] Scanning for unlocked files (this may take some time)
```

(continues on next page)

---

[262] https://console.aws.amazon.com/s3/

Fig. 13.9: The public GitHub repository with the DataLad dataset

(continued from previous page)

```
[INFO   ] Remote origin not usable by git-annex; setting annex-ignore
install(ok): /tmp/neuro-data-s3 (dataset)

$ cd neuro-data-s3
$ datalad get . -r
[INFO   ] Installing Dataset(/tmp/neuro-data-s3) to get /tmp/neuro-data-s3 recursively
get(ok): CHANGES (file) [from public-s3...]
get(ok): README (file) [from public-s3...]
get(ok): dataset_description.json (file) [from public-s3...]
get(ok): sub-01/anat/sub-01_T1w.nii (file) [from public-s3...]
get(ok): sub-01/func/sub-01_task-auditory_bold.nii (file) [from public-s3...]
get(ok): sub-01/func/sub-01_task-auditory_events.tsv (file) [from public-s3...]
get(ok): task-auditory_bold.json (file) [from public-s3...]
action summary:
get (ok: 7)
```

The results of running the code above show that DataLad could **install** the dataset correctly and **get** all annexed file content successfully from the `public-s3` sibling.

Congrats!

Advanced examples

When there is a lot to upload, automation is your friend. Below, we try to collect a few real world examples that go beyond toy examples. Do you have one and want to share it with the world as well? Please get in touch[263]!

---

[263] https://github.com/datalad-handbook/book/issues/new

Automated uploads of dataset hierarchies

The script below is a quick-and-dirty solution to the task of exporting a hierarchy of datasets to an S3 bucket. I needs to be invoked with three positional arguments, the path to the DATALAD SUPERDATASET, the S3 bucket name, and a prefix.

```bash
#!/bin/bash

set -eu

export PS4='> '
set -x

topds="$1"
bucket="$2"
prefix="$3"

# TEMP
srname="${bucket}5"

topdsfull=$PWD/$topds/

if ! git annex version | grep 8.2021 ; then
        echo "E: need recent git annex. check what you have"
        exit 1
fi

{ echo "$topdsfull"; datalad -f '{path}' subdatasets -r -d "$topds"; } | \
while read ds; do
        relds=$(relpath "$ds" "$topdsfull")
        fileprefix="$prefix/$relds/"
        fileprefix=$(python -c "import os,sys; print(os.path.normpath(sys.argv[1]))" "
→$fileprefix")
        echo $relds;
        (
                cd "$ds";
                # TODO: make sure that there is no ./ or // in fileprefix
                if ! git remote | grep -q "$srname"; then
                        git annex initremote --debug "$srname" \
                                type=S3 \
                                autoenable=true \
                                bucket=$bucket \
                                encryption=none \
                                exporttree=yes \
                                "fileprefix=$fileprefix/" \
                                host=s3.amazonaws.com \
                                partsize=1GiB \
                                port=80 \
                                "publicurl=https://s3.amazonaws.com/$bucket" \
                                public=yes \
                                versioning=yes
                fi
                git annex export --to "$srname" --jobs 6 master

        )
done
```

13.5 Walk-through: Git LFS as a special remote on GitHub

Some repository hosting services provide for-pay support for large files, and can thus be used as special remotes as well. GitHub and GitLab for example support Git Large File Storage[267] (Git LFS) for managing data files using Git. A free GitHub subscription allows up to 1GB of free storage and up to 1GB of bandwidth monthly[268]. As such, it might be sufficient for some use cases, and could be configured quite easily.

In order to store annexed dataset contents on GitHub, we need first to create a repository on GitHub:

```
$ datalad create-sibling-github test-github-lfs
.: github(-) [https://github.com/yarikoptic/test-github-lfs.git (git)]
'https://github.com/yarikoptic/test-github-lfs.git' configured as sibling 'github' for
↪<Dataset path=/tmp/test-github-lfs>
```

and then initialize a SPECIAL REMOTE of type `git-lfs`, pointing to the same GitHub repository:

```
$ git annex initremote github-lfs type=git-lfs url=https://github.com/yarikoptic/test-
↪github-lfs encryption=none embedcreds=no
```

If you would like to compress data in Git LFS, you need to take a detour via encryption during **git annex initremote** – this has compression as a convenient side effect. Here is an example initialization:

```
$ git annex initremote --force github-lfs type=git-lfs url=https://github.com/yarikoptic/
↪test-github-lfs encryption=shared
```

With this single step it becomes possible to transfer contents to GitHub:

```
$ git annex copy --to=github-lfs file.dat
copy file.dat (to github-lfs...)
ok
(recording state in git...)
```

and the entire dataset to the same GitHub repository:

```
$ datalad push --to=github
[INFO   ] Publishing <Dataset path=/tmp/test-github-lfs> to github
publish(ok): . (dataset) [pushed to github: ['[new branch]', '[new branch]']]
```

Because the special remote URL coincides with the regular remote URL on GitHub, `siblings` enable (as shown in the section *Walk-through: Dropbox as a special remote* (page 191)) will not even be necessary when datalad is installed from GitHub.

> ⚠️ **No drop from LFS**
>
> Unfortunately, it is impossible to **drop** contents from Git LFS: help.github.com/en/github/managing-large-files[269]
>
> ---
> [269] https://docs.github.com/en/github/managing-large-files/versioning-large-files/removing-files-from-git-large-file-storage#git-lfs-objects-in-your-repository

---

[267] https://github.com/git-lfs/git-lfs
[268] https://docs.github.com/en/github/managing-large-files/versioning-large-files/about-storage-and-bandwidth-usage

13.6 Walk-through: Dataset hosting on GIN

GIN[270] (G-Node infrastructure) is a free data management system designed for comprehensive and reproducible management of scientific data. It is a web-based repository store and provides fine-grained access control to share data. GIN builds up on GIT and GIT-ANNEX, and is an easy alternative to other third-party services to host and share your DataLad datasets[272]. It allows to share datasets and their contents with selected collaborators or making them publicly and anonymously available. *And even if you prefer to expose and share your datasets via GitHub, you can still use Gin to host your data* (page 214).
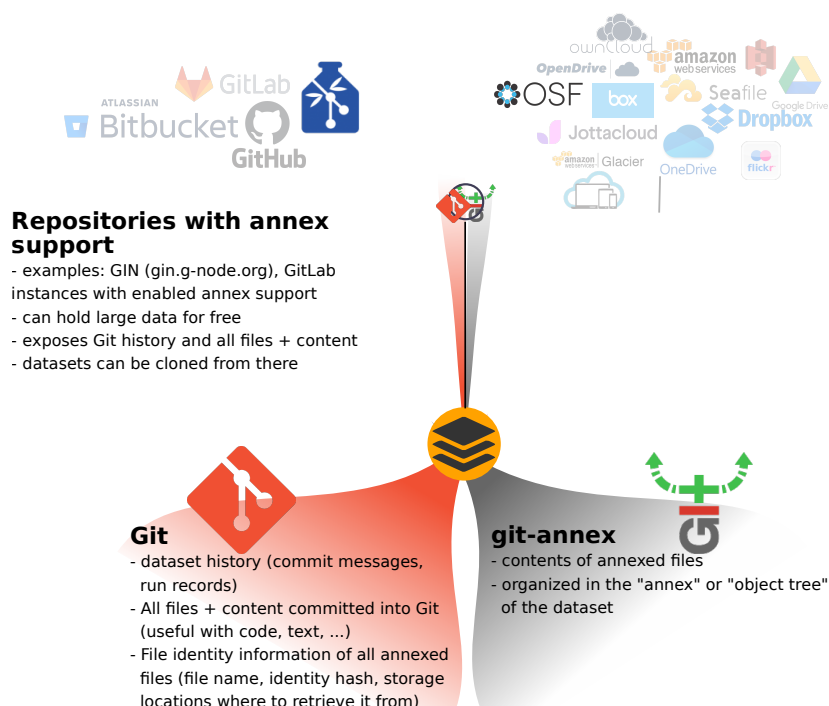
Fig. 13.10: Some repository hosting services such as Gin have annex support, and can thus hold the complete dataset. This makes publishing datasets very easy.

> ⚠ **Go further for dataset access from GIN**
>
> If you reached this section to find out how to access a DataLad dataset shared on Gin, please skip to the section *Sharing and accessing the dataset* (page 210).

Prerequisites

In order to use GIN for hosting and sharing your datasets, you need to

- register

- upload your public SSH KEY for SSH access

---

[270] https://gin.g-node.org/G-Node/Info/wiki

[272] GIN looks and feels similar to GitHub, and among a number advantages, it can assign a DOI to your dataset, making it cite-able. Moreover, its web interface[273] and client[274] are useful tools with a variety of features that are worthwhile to check out, as well.

[273] https://gin.g-node.org/G-Node/Info/wiki/WebInterface

[274] https://gin.g-node.org/G-Node/Info/wiki/GinUsageTutorial

- create an empty repository on GIN and publish your dataset to it

Once you have registered[271] an account on the GIN server by providing your e-mail address, affiliation, and name, and selecting a user name and password, you should upload your SSH KEY to allow SSH access (you can find an explanation of what SSH keys are and how you can create one in *this Findoutmore* (page 185) in the general section *Publishing datasets to Git repository hosting* (page 184)). To do this, visit the settings of your user account. On the left hand side, select the tab "SSH Keys", and click the button "Add Key":
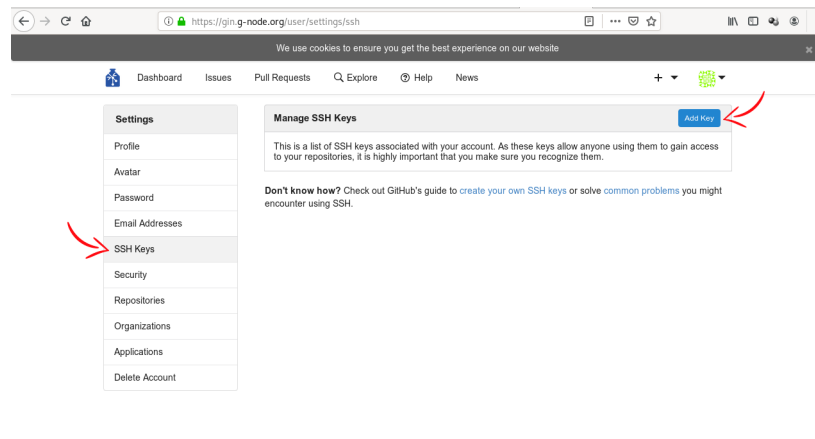
Fig. 13.11: Upload your SSH key to GIN

You should copy the contents of your public key file into the field labeled content, and enter an arbitrary but informative Key Name, such as "My private work station". Afterwards, you are done!

Publishing your dataset to GIN

To publish an existing dataset to GIN, create a new, empty repository on GIN first. Unlike with **datalad create-sibling-github** (that does this step automatically for you on GITHUB), this needs to be done via the web interface:

Afterwards, add this repository as a sibling of your dataset. To do this, use the **datalad siblings add** command and the SSH URL of the repository as shown below. Note that since this is the first time you will be connecting to the GIN server via SSH, you will likely be asked to confirm to connect. This is a safety measure, and you can type "yes" to continue:

```
$ datalad siblings add -d . \
  --name gin \
  --url git@gin.g-node.org:/adswa/DataLad-101.git


The authenticity of host 'gin.g-node.org (141.84.41.219)' can't be established.
ECDSA key fingerprint is SHA256:E35RRG3bhoAm/WD+0dqKpFnxJ9+yi0uUiFLi+H/lkdU.
Are you sure you want to continue connecting (yes/no)? yes
[INFO   ] Failed to enable annex remote gin, could be a pure git or not accessible
[WARNING] Failed to determine if gin carries annex.
.: gin(-) [git@gin.g-node.org:/adswa/DataLad-101.git (git)]
```

Afterwards, you can publish your dataset with **datalad push**. As the repository on GIN supports a dataset annex, there is no publication dependency to an external data hosting service

---

[271] https://gin.g-node.org/user/sign_up

Fig. 13.12: Create a new repository on Gin using the web interface.

necessary, and the dataset contents stored in Git and in git-annex are published to the same place:

```
$ datalad push --to gin
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Transfer data
copy(ok): books/TLCL.pdf (file) [to gin...]
copy(ok): books/bash_guide.pdf (file) [to gin...]
copy(ok): books/byte-of-python.pdf (file) [to gin...]
copy(ok): books/progit.pdf (file) [to gin...]
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
[INFO] Start resolving deltas
publish(ok): . (dataset) [refs/heads/git-annex->gin:refs/heads/git-annex 6714aa0..455829a]
publish(ok): . (dataset) [refs/heads/master->gin:refs/heads/master [new branch]]
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101)
```

If you refresh the GIN web interface afterwards, you will find all of your dataset – including annexed contents! – on GIN. What is especially cool is that the GIN web interface (unlike GITHUB) can even preview your annexed contents.

Sharing and accessing the dataset

Once your dataset is published, you can point collaborators and friends to it.

If it is a **public** repository, retrieving the dataset and getting access to all published data contents (in a read-only fashion) is done by cloning the repository's `https` url. This does not require a user account on Gin.
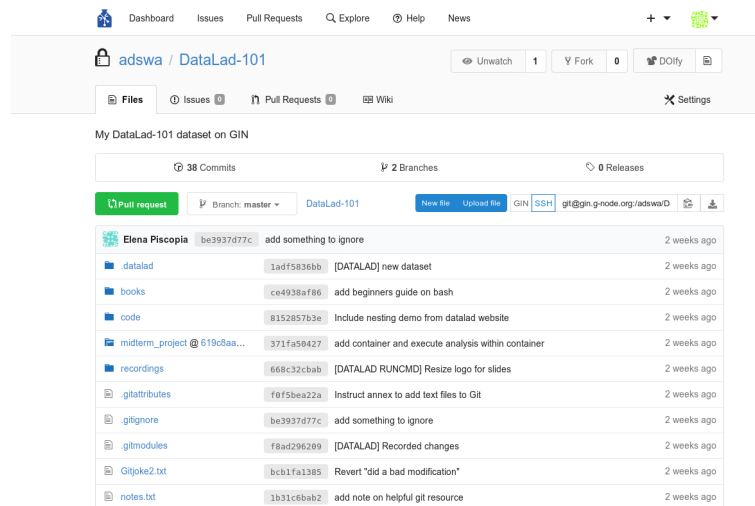
Fig. 13.13: A published dataset in a Gin repository at gin.g-node.org.

> ⚠️ **Take the URL in the browser, not the copy-paste URL**
>
> Please note that you need to use the browser URL of the repository, not the copy-paste URL on the upper right hand side of the repository if you want to get anonymous HTTPS access! The two URLs differ only by a `.git` extension:
> - Browser bar: `https://gin.g-node.org/<user>/<repo>`
> - Copy-paste "HTTPS clone": `https://gin.g-node.org/<user>/<repo>.git`
>
> A dataset cloned from `https://gin.g-node.org/<user>/<repoy>.git`, however, can not retrieve annexed files!

```
$ datalad clone https://gin.g-node.org/adswa/DataLad-101
[INFO] Cloning dataset to Dataset(/home/me/dl-101/clone_of_dl-101/DataLad-101)
[INFO] Attempting to clone from https://gin.g-node.org/adswa/DataLad-101 to /home/me/dl-
↪101/clone_of_dl-101/DataLad-101
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/clone_of_dl-101/DataLad-101)
[INFO] scanning for unlocked files (this may take some time)
install(ok): /home/me/dl-101/clone_of_dl-101/DataLad-101 (dataset)
```

Subsequently, **datalad get** calls will be able to retrieve all annexed file contents that have been published to the repository.

If it is a **private** dataset, cloning the dataset from Gin requires a user name and password for anyone you want to share your dataset with. The "Collaboration" tab under Settings lets you set fine-grained access rights, and it is possible to share datasets with collaborators that are not registered on GIN with provided Guest accounts. If you are unsure if your dataset is private, *this find-out-more shows you how to find out* (page 212). In order to get access to annexed contents, cloning *requires* setting up an SSH key as detailed above, and cloning via the SSH url:
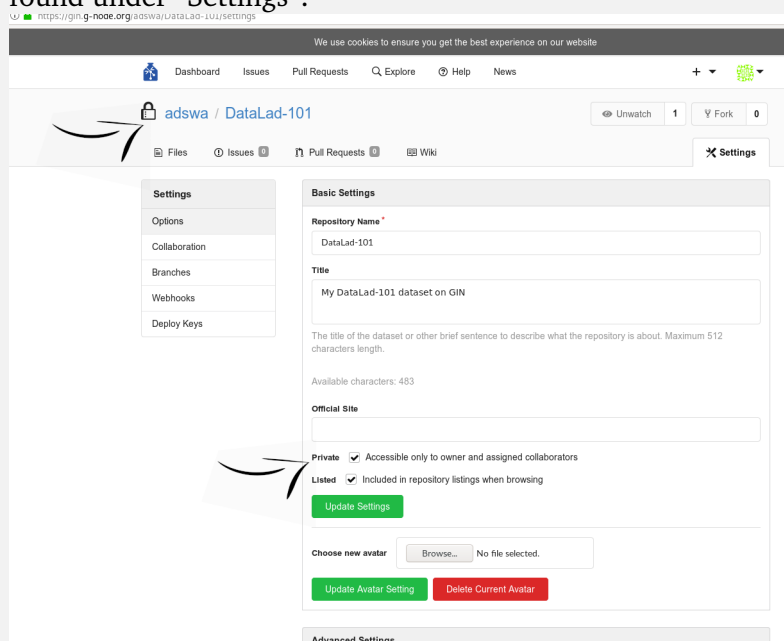
```
$ datalad clone git@gin.g-node.org:/adswa/DataLad-101.git
```

Likewise, in order to publish changes back to a Gin repository, the repository needs to be cloned via its SSH url.

> ### 💡 M13.5 How do I know if my repository is private?
>
> Private repos are marked with a lock sign. To make it public, untick the "Private" box, found under "Settings":
>
>

Subdataset publishing

Just as the input subdataset `iris_data` in your published `midterm_project` was referencing its source on GITHUB, the `longnow` subdataset in your published `DataLad-101` dataset directly references the original dataset on GITHUB. If you click onto `recordings` and then `longnow` in GIN's webinterface, you will be redirected to the podcast's original dataset.

The subdataset `midterm_project`, however, is not successfully referenced. If you click on it, you would get to a 404 Error page. The crucial difference between this subdataset and the `longnow` dataset is its entry in the `.gitmodules` file of `DataLad-101`:

```
$ cat .gitmodules
[submodule "recordings/longnow"]
        path = recordings/longnow
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
[submodule "midterm_project"]
        path = midterm_project
        url = ./midterm_project
        datalad-id = e5a3d370-223d-11ea-af8b-e86a64c8054c
```

While the longnow subdataset is referenced with a valid URL to GitHub, the midterm project's URL is a relative path from the root of the superdataset. This is because the `longnow` subdataset was installed with **datalad clone -d .** (that records the source of the subdataset), and the `midterm_project` dataset was created as a subdataset with **datalad create**

-d . midterm_project. Since there is no repository at `https://gin.g-node.org/<USER>/`
`DataLad-101/midterm_project` (which this submodule entry would resolve to), accessing the
subdataset fails.

However, since you have already published this dataset (to GitHub), you could update the
submodule entry and provide the accessible GitHub URL instead. This can be done via the
`set-property <NAME> <VALUE>` option of **datalad subdatasets**[275] (replace the URL shown here
with the URL your dataset was published to – likely, you only need to change the user name):

```
$ datalad subdatasets --contains midterm_project \
  --set-property url https://github.com/adswa/midtermproject
subdataset(ok): midterm_project (dataset)
```

```
$ cat .gitmodules
[submodule "recordings/longnow"]
        path = recordings/longnow
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
        datalad-url = https://github.com/datalad-datasets/longnow-podcasts.git
[submodule "midterm_project"]
        path = midterm_project
        url = https://github.com/adswa/midtermproject
        datalad-id = fea0e9c7-7932-41d5-8d57-97ac9ac6755a
```

Handily, the **datalad subdatasets** command saved this change to the `.gitmodules` file auto-
matically and the state of the dataset is clean:

```
$ datalad status
nothing to save, working tree clean
```

Afterwards, publish these changes to gin and see for yourself how this fixed the problem:

```
$ datalad push --to gin
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Transfer data
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
publish(ok): . (dataset) [refs/heads/master->gin:refs/heads/master b1c8ba0..c35b6c1]
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101)
```

If the subdataset was not published before, you could publish the subdataset to a location of
your choice, and modify the `.gitmodules` entry accordingly.

---

[275] Alternatively, you can configure the siblings url with **git config**:

```
$ git config -f .gitmodules --replace-all  submodule.midterm_project.url https://github.com/adswa/
↪midtermproject
```

Remember, though, that this command modifies `.gitmodules` *without* an automatic, subsequent **save**, so that you
will have to save this change manually.

Using Gin as a data source behind the scenes

Even if you do not want to point collaborators to yet another hosting site but want to be able to expose your datasets via services they use and know already (such as GitHub or GitLab), Gin can be very useful: You can let Gin perform data hosting in the background by using it as an "autoenabled data source" that a dataset SIBLING (even if it is published to GitHub or GitLab) can retrieve data from. You will need to have a Gin account and SSH key setup, so please take a look at the first part of this section if you do not yet know how to do this.

Then, follow these steps:

- First, create a new repository on Gin (see step by step instructions above).

- In your to-be-published dataset, add this repository as a sibling, this time setting *–url* and *–pushurl* arguments explicitly. Make sure to configure a SSH URL as a `--pushurl` but a HTTPS URL as a url. Please also note that the HTTPS URL written after `--url` DOES NOT have the `.git` suffix. Here is the command:

```
$ datalad siblings add \
 -d . \
 --name gin \
 --pushurl git@gin.g-node.org:/studyforrest/aggregate-fmri-timeseries.git \
 --url https://gin.g-node.org/studyforrest/aggregate-fmri-timeseries \
```

- Locally, run `git config --unset-all remote.gin.annex-ignore` to prevent GIT-ANNEX from ignoring this new dataset

- Push your data to the repository on Gin (`datalad push --to gin`). This pushes the actual state of the repository, including content, but also adjusts the GIT-ANNEX configuration.

- Configure this sibling as a "common data source". Use the same name as previously in `--name` (to indicate which sibling you are configuring) and give a new, different, name after `--as-common-datasrc`:

```
$ datalad siblings configure \
    --name gin \
    --as-common-datasrc gin-src
```

- Push to the repository on Gin again (`datalad push --to gin`) to make the configuration change known to the Gin sibling.

- Publish your dataset to GitHub/GitLab/..., or update an existing published dataset (`datalad push`)

Afterwards, **datalad get** retrieves files from Gin, even if the dataset has been cloned from GitHub.

> **G13.2 siblings as a common data source**
>
> The argument `as-common-datasrc <name>` configures a sibling as a common data source – in technical terms, as an auto-enabled git-annex special remote.

13.7 Built-in data export

Apart from flexibly configurable special remotes that allow publishing annexed content to a variety of third party infrastructure, DataLad also has some built-in support for "exporting" data to other services. This usually means that a static snapshot of your dataset and its files are shared in archives or collections of files. While an export of a dataset looses some of the advantages that a DataLad dataset has, for example a transparent version history, it can be a fast and simple way to make the most recent version of your dataset available or archived.

One example is the command **export-archive**. Running this command creates a `.tar.gz` file with the content of your dataset. This compressed archive can be uploaded to any data hosting portal manually. This moves data out of version control and decentralized tracking, and essentially "throws it over the wall" - while your data (also the annexed data) will be available for download from where you share it, none of the special features a DataLad dataset provides will be available, such as its history or configurations.

Another example is **export-to-figshare**. Figshare[276] is an online open access repository where researchers can preserve and share their research outputs, including figures, datasets, or images - and thus everything that could potentially be managed in a Datalad dataset. Running **datalad export-to-figshare** allows you to publish the dataset as a snapshot. Note that this requires a free account on Figshare, and the generation of an access token[277] for authentication. An interactive prompt will ask you to supply authentication credentials, and guide you through the process of creating a new article.

```
$ datalad export-to-figshare
    [INFO   ] Exporting current tree as an archive under /home/me/DataLad-101 since␣
→figshare does not support directories
    [INFO   ] Uploading /home/me/datalad_b1cbbaa9-dd5c-473e-8092-e911021f33cb.zip to␣
→figshare
    Article
    Would you like to create a new article to upload to?  If not - we will list existing␣
→articles (choices: yes, no): yes

    New article
    Please enter the title (must be at least 3 characters long). [abcd#b1cbbaa9-dd5c-
→473e-8092-e911021f33cb]: my-cool-dataset

    [INFO   ] Created a new (private) article 16676764 at https://figshare.com/account/
→articles/16676764. Please visit it, enter additional meta-data and make public
    [INFO   ] 'Registering' /home/me/datalad_b1cbbaa9-dd5c-473e-8092-e911021f33cb.zip␣
→within annex
    [INFO   ] Adding URL https://ndownloader.figshare.com/files/30876682 for it
    [INFO   ] Registering links back for the content of the archive
    [INFO   ] Adding content of the archive /home/me/datalad_b1cbbaa9-dd5c-473e-8092-
→e911021f33cb.zip into annex AnnexRepo(/home/me/DataLad-101)
    [INFO   ] Initiating special remote datalad-archives
    [INFO   ] Finished adding /home/me/datalad_b1cbbaa9-dd5c-473e-8092-e911021f33cb.zip:␣
→Files processed: 4, removed: 4, +git: 2, +annex: 2
    [INFO   ] Removing generated and now registered in annex archive
    export_to_figshare(ok): Dataset(/home/me/DataLad-101) [Published archive https://
→ndownloader.figshare.com/files/30876682]
```

The screenshot below shows how the `DataLad-101` dataset looks like in exported form:

---

[276] https://figshare.com/
[277] https://figshare.com/account/applications

You could then extend the dataset with metadata, obtain a DOI[278] for it and make it citable, and point others to it in order to download it as an archive of files.

Beyond this, as the command `export-archive` is used by it to prepare content for upload to Figshare, annexed files also will be annotated as available from the archive on Figshare using `datalad-archive` special remote. As a result, if you publish your Figshare dataset and share your DataLad dataset on a repository hosting service without support for annexed files, users will still be able to fetch content from the tarball shared on Figshare.

```
$ datalad siblings
 .: here(+) [git]
 .: datalad-archives(+) [datalad-archives]
```

13.8 Keeping (some) dataset contents private

Datasets can contain information that you don't want to share with others. Maybe the collection of pictures from your team-building event also contains those after-hour photos where you drunkenly kidnapped a tram. Or you are handling data with strict privacy requirements, such as patient data or medical imaging files. Whatever it may be, this short section summarizes strategies that help you to ensure to private information is not leaked, even when you publicly share datasets that contain it.

Strategy 1: Never save private information to Git

The most important strategy to keep in mind in handling datasets with potentially sensitive information is to **never save sensitive information into Git**. **NEVER**. Saving sensitive information into a dataset or Git repository that you intend to share is the equivalent of including your account password as an attachment to every email you write – you don't necessarily point out that there is private information, but it lies around for everyone to accidentally find. Once a file with sensitive contents has been saved in the version history, sharing this dataset may accidentally expose the sensitive information even if it has been removed in the most recent version – the transparent revision history of a dataset allows to simply restore the file.

Thus, make sure to always manage sensitive files with GIT-ANNEX, even if the file is just a small text file. Having the file annexed allows you to specifically not share its contents, even when you make your dataset publicly available. However, it is highly important to realize that while annexed file's *contents* are not saved into Git, annex file's *names* are. If private information such as a medical patients non-anonymized ID or other potentially identifying information becomes a part of the file name, this information is exposed in the Git history of the dataset. Keep in mind that this applies even if you renamed the file.

> **M13.6 Help! I accidentally saved sensitive information to Git!**
>
> The only lasting way to remove contents from the dataset history completely is to substantially rewrite the dataset's history via tools such as `git-filter-repo` or `git filter-branch`, two very dangerous and potentially destructive operations. If you ever need to go there, the advanced section *Fixing up too-large datasets* (page 328) contains a paragraph on "Getting contents out of Git".

---

[278] https://www.doi.org/driven_by_DOI.html

Strategy 2: Restrict access via third party service or file system permissions

When you have a dataset and only authorized actors should be allowed to access it, it is possible to set access restrictions simply via choice of (third party) storage permissions. When it is an access restricted dataset on shared infrastructure, for example a scientific dataset that only researchers who signed a data usage agreement should have access to, it could suffice to create specific Unix groups[279] with authorized users, and give only those groups the necessary permissions. Depending on what permissions are set, unauthorized actors would not be able to retrieve file contents, or be able to clone the dataset at all.

The ability of repository hosting services to make datasets private and only allow select collaborators access is yet another method of keeping complete datasets as private as necessary, even though you should think twice on whether or not you should host sensitive repositories at all on these services.

One method to exert potentially fine-grained access control over file contents is via choice of (third party) hosting service for some or all annexed file contents. If you chose a service only selected people have access to, and publish annexed contents exclusively there, then only those selected people can perform a successful **datalad get**. For example, when it is a dataset with content hosted on third party cloud storage such as S3 buckets, permission settings in the storage locations would allow data providers to specify or limit who is able to retrieve the file contents. An example for this is the usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433), where file contents from the human connectome project can only be retrieved when a user has obtained the necessary credentials first.

Strategy 3: Selective publishing

If it is individual files that you do not want to share, you can selectively publish the contents of all files you want others to have, and withhold the data of the files you do not want to share. This can be done by providing paths to the data that should be published, or a git-annex-wanted[280] configuration and the --data auto option.

Let's say you have a dataset with three files:

- experiment.txt
- subject_1.dat
- subject_2.data

Consider that all of these files are annexed. While the information in experiment.txt is fine for everyone to see, subject_1.dat and subject_2.dat contain personal and potentially identifying data that can not be shared. Nevertheless, you want collaborators to know that these files exist. By publishing only the file contents of experiment.txt with

```
$ datalad push --to github experiment.txt
```

only meta data about file availability of subject_1.dat and subject_2.dat exists, but as these files' annexed data is not published, a **datalad get** will fail. Note, though, that **push** will publish the complete dataset history (unless you specify a commit range with the --since option – see the manual[281] for more information).

---

[279] https://en.wikipedia.org/wiki/Group_identifier
[280] https://git-annex.branchable.com/git-annex-wanted/
[281] http://docs.datalad.org/en/latest/generated/man/datalad-push.html

13.9 Overview: The datalad push command

Previous sections on publishing DataLad datasets have each shown you crucial aspects of the functions of dataset publishing with **datalad push**. This section wraps them all together.

> ⚠️ **push availability**
>
> **datalad push** requires DataLad version 0.13.0 or higher. Older DataLad versions need to use the **datalad publish** command. For details into datalad publish, please check out the *find-out-more on the difference between the two commands* (page 223) at the end of this page.

The general overview

datalad push is the command to turn to when you want to publish datasets. It is capable of publishing all dataset content, i.e., files stored in GIT, and files stored with GIT-ANNEX, to a known dataset SIBLING.

> **G13.3 Push internals**
>
> The **datalad push** uses git push, and git annex copy under the hood. Publication targets need to either be configured remote Git repositories, or git-annex special remotes (if they support data upload).

In order to publish a dataset, the dataset needs to have a sibling to push to. This, for instance, can be a GITHUB, GITLAB, or GIN repository, but it can also be a Remote Indexed Archive (RIA) store for backup or storage of datasets[286], or a regular clone.

> 💡 **M13.7 all of the ways to configure siblings**
>
> - Add an existing repository as a sibling with the **datalad siblings** command. Here are common examples:
>
>   ```
>   # to a remote repository
>   $ datalad siblings add --name github-repo --url <url.to.github>
>   # to a local path
>   $ datalad siblings add --name local-sibling --url /path/to/sibling/ds
>   # to a clone on an SSH-accessible machine
>   $ datalad siblings add --name server-sibling --url [user@]hostname:/path/to/
>   ↪sibling/ds
>   ```
>
> - Create a sibling on an external hosting service from scratch, right from within your repository: This can be done with the commands **create-sibling-github** (for GitHub) or **create-siblings-gitlab** (for GitLab), or **create-sibling-ria** (for a remote indexed archive dataset store[286]). Note that **create-sibling-ria** can add an existing store as a sibling or create a new one from scratch.
> - Create a sibling on a local or SSH accessible Unix machine with **datalad create-sibling** (datalad-create-sibling manual).

---

[286] RIA siblings are filesystem-based, scalable storage solutions for DataLad datasets. You can find out more about them in the section *Remote Indexed Archives for dataset storage and backup* (page 294).

In order to publish dataset content, DataLad needs to know to which sibling content shall be pushed. This can be specified with the `--to` option directly from the command line:

```
$ datalad push --to <sibling>
```

If you have more than one BRANCH in your dataset, note that a **datalad push** command will by default update only the current branch. If updating multiple branches is relevant for your workflow, please check out the *find-out-more about this* (page 221).

By default, **push** will make the last saved state of the dataset available. Consequently, if the sibling is in the same state as the dataset, no push is attempted. Additionally, **push** will attempt to automatically decide what type of dataset contents are going to be published. With a sibling that has a SPECIAL REMOTE configured as a PUBLICATION DEPENDENCY, or a sibling that contains an annex (such as a Gin repository or a REMOTE INDEXED ARCHIVE (RIA) STORE), both the contents stored in Git (i.e., a dataset's history) as well as file contents stored in git-annex will be published unless dataset configurations overrule this. Alternatively, one can enforce particular operations or push a subset of dataset contents. For one, when specifying a path in the **datalad push** command, only data or changes for those paths are considered for a push. Additionally, one can select a particular mode of operation with the `-data` option. Several different modes are possible:

- `nothing`: With this option, annexed contents are not published. This means that the sibling will have information on the annexed files' names, but file contents will not be available, and thus `datalad get` calls in the sibling would fail.

- `anything`: Transfer all annexed contents.

- `auto`: With this option, the decision which data is transferred is based on configurations that can determine rules on a per-file and per-sibling level. On a technical level, the `git annex copy` call to publish file contents is called with its `--auto` option. With this option, only data that satisfies specific git-annex configurations gets transferred. Those configurations could be `numcopies` settings (the number of copies available at different remotes), or `wanted` settings (preferred contents for a specific remote), and need to be created by a user[287] with git-annex commands. If you have files you want to keep private, or do not need published, these configurations are very useful.

- `auto-if-wanted` (Default): Unless a `wanted` or `numcopies` configuration exists in the dataset, all content are published. Should a `wanted` or `numcopies` configuration exist, the command enables `--auto` in the underlying `git annex copy` call.

Beyond different modes of transferring data, the `-f/--force` option allows to force specific publishing operations with three different modes. Be careful when using it, as its modes possibly overrule safety protections or optimizations:

- `checkdatapresent`: With this option, the underlying `git annex copy` call to publish file contents is invoked without a `--fast` option. Usually, the `--fast` option increases the speed of the operation, as it disables a check whether the sibling already has content. This however, might skip copying content in some cases. Therefore, `--force datatransfer` is a slower, but more fail-safe option to publish annexed file contents.

- `gitpush`: This option triggers a `git push --force`. Be very careful using this option! If the changes on the dataset conflict with the changes that exist in the sibling, the changes

---

[287] For information on the `numcopies` and `wanted` settings of git-annex see its documentation at git-annex.branchable.com/git-annex-wanted/[288] and git-annex.branchable.com/git-annex-numcopies/[289].

[288] https://git-annex.branchable.com/git-annex-wanted/

[289] https://git-annex.branchable.com/git-annex-numcopies/

in the sibling will be overwritten.

- all: The final mode, all, combines all force modes – thus attempting to really get your dataset contents published by any means.

datalad push can publish available subdatasets recursively if the -r/--recursive flag is specified. Note that this requires that all subdatasets that should be published have sibling names identical to the sibling specified in the top-level **push** command, or that appropriate default publication targets are configured throughout the dataset hierarchy.

> **M13.8 Pushing more than the current branch**
>
> If you have more than one BRANCH in your dataset, a **datalad push --to <sibling>** will by default only push the current BRANCH, *unless* you provide configurations that alter this default. Here are two ways in which this can be achieved:
>
> **Option 1:** Setting the push.default configuration variable from simple (the default) to matching will configure the dataset such that **push** pushes *all* branches to the sibling. A concrete example: On a dataset level, this can be done using
>
> ```
> $ git config --local push.default matching
> ```
>
> **Option 2:** Tweaking the default push refspec[282] for the dataset allows to select a range of branches that should be pushed. The link above gives a thorough introduction into the refspec. For a hands-on example, consider how it is done for the published DataLad-101 dataset[283]:
>
> The published version of the handbook is known to the local handbook dataset as a REMOTE called public, and each section of the book is identified with a custom branch name that corresponds to the section name. Whenever an update to the public dataset is pushed, apart from pushing only the master branch, all branches starting with the section identifier sct are pushed automatically as well. This configuration was achieved by specifying these branches (using GLOBBING with *) in the push specification of this REMOTE:
>
> ```
> $ git config --local remote.public.push 'refs/heads/sct*'
> ```
>
> ---
> [282] https://git-scm.com/book/en/v2/Git-Internals-The-Refspec
> [283] https://github.com/datalad-handbook/DataLad-101

Pushing errors

If you are unfamiliar with Git, please be aware that cloning a dataset to a different place and subsequently pushing to it can lead to Git error messages if changes are pushed to a currently checked out BRANCH of the sibling (in technical Git terms: When pushing to a checked-out branch of a non-bare repository remote). As an example, consider what happens if we attempt a **datalad push** to the sibling roommate that we created in the chapter *Collaboration* (page 92):

```
$ datalad push --to roommate
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Transfer data
copy(ok): books/TLCL.pdf (file) [to roommate...]
copy(ok): books/bash_guide.pdf (file) [to roommate...]
```

<div align="right">(continues on next page)</div>

```
copy(ok): books/byte-of-python.pdf (file) [to roommate...]
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
[INFO] Start resolving deltas
[INFO] Finished
publish(ok): . (dataset) [refs/heads/git-annex->roommate:refs/heads/git-annex dc458de..
↪2f9d5f3]
[ERROR] refs/heads/master->roommate:refs/heads/master [remote rejected] (branch is␣
↪currently checked out) [publish(/home/me/dl-101/DataLad-101)]
publish(error): . (dataset) [refs/heads/master->roommate:refs/heads/master [remote␣
↪rejected] (branch is currently checked out)]
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101)
```

Publishing fails with the error message `[remote rejected] (branch is currently checked out)`. This can be prevented with configuration settings[284] in Git versions 2.3 or higher, or by pushing to a branch of the sibling that is currently not checked-out. For more information on this, and other error messages during push, please checkout the section *How to get help* (page 258).

13.10 Summary

Without access to the same computational infrastructure, you can share your DataLad datasets with friends and collaborators by leveraging third party services. DataLad integrates well with a variety of free or commercial services, and with many available service options this gives you freedom in deciding where you store your data and thus who can get access.

- An easy, free, and fast option is GIN[290], a web-based repository store for scientific data management. If you are registered and have SSH authentication set up, you can create a new, empty repository, add it as a sibling to your dataset, and publish all dataset contents – including annexed data, as GIN supports repositories with an annex.

- Other repository hosting services such as GitHub and GitLab[292] do not support an annex. If a dataset is shared via one of those platforms, annexed data needs to be published to an external data store. The published dataset stores information about where to obtain annexed file contents from such that a **datalad get** works.

- The external data store can be any of a variety of third party hosting providers. To enable data transfer to and from this service, you (may) need to configure an appropriate SPECIAL REMOTE, and configure a publication dependency. The section *Beyond shared infrastructure* (page 177) walked you through how this can be done with Dropbox[291].

- The `--data` and `--force` options of **datalad push** allows to override automatic decision making on to-be-published contents. If it isn't specified, DataLad will attempt to figure out

---

[284] https://github.blog/2015-02-06-git-2-3-has-been-released/

[290] https://gin.g-node.org

[292] Older versions of GITLAB provide a git-annex configuration, but it is disabled by default, and to enable it you would need to have administrative access to the server and client side of your GitLab instance. Find out more here[293].

[293] https://docs.gitlab.com/12.10/ee/administration/git_annex.html

[291] https://dropbox.com

> 💡 **M13.9 On the datalad publish command**
>
> Starting with DataLad version `0.13.0`, **datalad push** was introduced and became an alternative to **datalad publish**, which will be removed in a future DataLad release.
>
> By default, **datalad publish** publishes the last saved state of the dataset (i.e., its Git history) to a specified sibling:
>
> ```
> $ datalad publish --to <sibling>
> ```
>
> Like **push**, it supports recursive publishing across dataset hierarchies (if all datasets have appropriately configured default publication targets or identical sibling names) with the `-r`/`--recursive` flag, and it supports the `--since` option.
>
> Main differences to **push** lie in **publish**s `--transfer-data` option that can be specified with either `all`, `auto` or `none` and determines whether and how annexed contents should be published if the sibling carries an annex: `none` will transfer only Git history and no annexed data, `auto` relies on configurations of the sibling, and `all` will publish all annexed contents.
>
> By default, when using a plain `datalad publish --to <sibling>` with no path specification or `--transfer-data` option, **publish** will be used in `auto` mode. In practice, this default will most likely lead to the same outcome as when specifying `none`: only your datasets history, but no annexed contents will be published. On a technical level, the `auto` option leads to adding `auto` to the underlying `git annex copy` command, which in turn publishes annexed contents based on the git-annex preferred content configuration[285] of the sibling.
>
> In order to publish all annexed contents, one needs to specify `--transfer-data all`. Alternatively, adding paths to the `publish` call will publish the specified annexed content (unless `--transfer-data none` is explicitly added). As yet another alternative, one needs to add the same configuration for GIT-ANNEX that the option `--auto` of **push** need.
>
> ---
> [285] https://git-annex.branchable.com/git-annex-preferred-content/

itself which and how dataset contents shall be published. With a path to files, directories, or subdatasets you can also publish only selected contents' data.

Now what can I do with it?

Finally you can share datasets and their annexed contents with others without the need for a shared computational infrastructure. It remains your choice where to publish your dataset to – considerations of data access, safety, or potential costs will likely influence your choice of service.

HELP YOURSELF
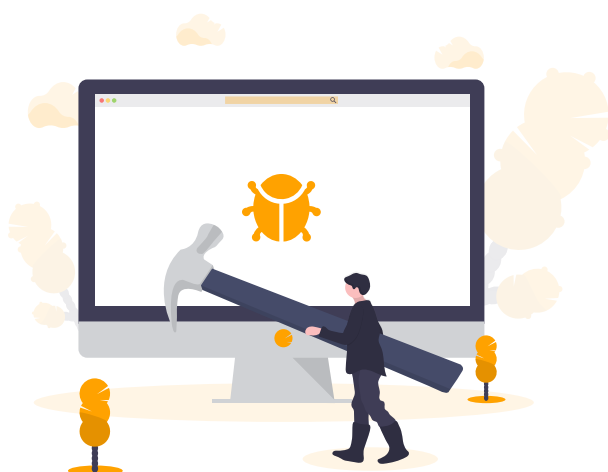
14.1 What to do if things go wrong

After all of the DataLad-101 lectures and tutorials so far, you really begin to appreciate the pre-crafted examples and tasks the handbook provides. "Nothing really goes wrong, and if so, it's intended", you acknowledge. "But how does this prepare me for life after the course? I've seen a lot of different errors and know many caveats and principles already, but I certainly will mess something up at one point. How can I get help, or use the history of the dataset to undo what I screwed up? Also, I'm not sure whether I know what I can and can not do with the files inside of my dataset... What if I would like to remove one, for example?"

Therefore, this upcoming chapter is a series of tutorials about common file system operations, interactions with the history of datasets, and how to get help after errors.

14.2 Miscellaneous file system operations

With all of the information about symlinks and object trees, you might be reluctant to perform usual file system managing operations, such as copying, moving, renaming or deleting files or directories with annexed content.

If I renamed one of those books, would the symlink that points to the file content still be correct? What happens if I'd copy an annexed file? If I moved the whole books/ directory? What if I moved all of DataLad-101 into a different place on my computer? What if renamed the whole superdataset? And how do I remove a file, or directory, or subdataset?

Therefore, there is an extra tutorial offered by the courses' TA today, and you attend. There is no better way of learning than doing. Here, in the safe space of the `DataLad-101` course, you can try out all of the things you would be unsure about or reluctant to try on the dataset that contains your own, valuable data.

Below you will find common questions about file system management operations, and each question outlines caveats and solutions with code examples you can paste into your own terminal. Because these code snippets will add many commits to your dataset, we're cleaning up within each segment with common git operations that manipulate the datasets history – be sure to execute these commands as well (and be sure to be in the correct dataset).

Renaming files

Let's try it. In Unix, renaming a file is exactly the same as moving a file, and uses the `mv` command.

```
$ cd books/
$ mv TLCL.pdf The_Linux_Command_Line.pdf
$ ls -lah
total 24K
drwxr-xr-x 2 adina adina 4.0K Jul 29 16:24 .
drwxr-xr-x 8 adina adina 4.0K Jul 29 16:24 ..
lrwxrwxrwx 1 adina adina  131 Jan 19  2009 bash_guide.pdf -> ../.git/annex/objects/WF/Gq/
↪MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--
↪0ab2c121bcf68d7278af266f6a399c5f.pdf
lrwxrwxrwx 1 adina adina  131 Jun 16  2020 byte-of-python.pdf -> ../.git/annex/objects/z1/
↪Q8/MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf/MD5E-s4208954--
↪ab3a8c2f6b76b18b43c5949e0661e266.pdf
lrwxrwxrwx 1 adina adina  133 Jun 29  2019 progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-
↪s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--
↪05cd7ed561d108c9bcf96022bc78a92c.pdf
lrwxrwxrwx 1 adina adina  131 Jan 28  2019 The_Linux_Command_Line.pdf -> ../.git/annex/
↪objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf
```

Try to open the renamed file, e.g., with `evince The_Linux_Command_Line.pdf`. This works!

But let's see what changed in the dataset with this operation:

```
$ datalad status
untracked: /home/me/dl-101/DataLad-101/books/The_Linux_Command_Line.pdf (symlink)
  deleted: /home/me/dl-101/DataLad-101/books/TLCL.pdf (symlink)
```

We can see that the old file is marked as `deleted`, and simultaneously, an `untracked` file appears: the renamed PDF.

While this might appear messy, a `datalad save` will clean all of this up. Therefore, do not panic if you rename a file, and see a dirty dataset status with deleted and untracked files – `datalad save` handles these and other cases really well under the hood.

```
$ datalad save -m "rename the book"
delete(ok): books/TLCL.pdf (file)
add(ok): books/The_Linux_Command_Line.pdf (file)
save(ok): . (dataset)
action summary:
```

```
  add (ok: 1)
  delete (ok: 1)
  save (ok: 1)
```

The **datalad save** command will identify that a file was renamed, and will summarize this nicely in the resulting commit:

```
$ git log -n 1 -p
commit 6a97027f7cea0afd1f20e333d9d69c0942a8c48b
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:00 2021 +0200

    rename the book

diff --git a/books/TLCL.pdf b/books/The_Linux_Command_Line.pdf
similarity index 100%
rename from books/TLCL.pdf
rename to books/The_Linux_Command_Line.pdf
```

Note that **datalad save** commits all modifications when it's called without a path specification, so any other changes will be saved in the same commit as the rename. If there are unsaved modifications you do not want to commit together with the file name change, you could give both the new and the deleted file as a path specification to **datalad save**, even if it feels unintuitive to save a change that is marked as a deletion in a **datalad status**:

```
datalad save -m "rename file" oldname newname
```

Alternatively, there is also a way to save the name change only using Git tools only, outlined in the following hidden section. If you are a Git user, you will be very familiar with it.

> ### M14.1 Renaming with Git tools
>
> Git has built-in commands that provide a solution in two steps.
> If you have followed along with the previous **datalad save**, let's revert the renaming of the the files:
>
> ```
> $ git reset --hard HEAD~1
> $ datalad status
> HEAD is now at 1fae7c2 add container and execute analysis within container
> nothing to save, working tree clean
> ```
>
> Now we're checking out how to rename files and commit this operation using only Git: A Git-specific way to rename files is the git mv command:
>
> ```
> $ git mv TLCL.pdf The_Linux_Command_Line.pdf
> ```
>
> ```
> $ datalad status
>     added: /home/me/dl-101/DataLad-101/books/The_Linux_Command_Line.pdf (file)
>   deleted: /home/me/dl-101/DataLad-101/books/TLCL.pdf (file)
> ```
>
> We can see that the old file is still seen as "deleted", but the "new", renamed file is "added". A git status displays the change in the dataset a bit more accurately:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    TLCL.pdf -> The_Linux_Command_Line.pdf
```

Because the **git mv** places the change directly into the staging area (the *index*) of Git[297], a subsequent `git commit -m "rename book"` will write the renaming – and only the renaming – to the dataset's history, even if other (unstaged) modifications are present.

```
$ git commit -m "rename book"
[master 6dbccb8] rename book
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename books/{TLCL.pdf => The_Linux_Command_Line.pdf} (100%)
```

---

[297] If you want to learn more about the Git-specific concepts of *worktree*, *staging area/index* or *HEAD*, the upcoming section *Back and forth in time* (page 245) will talk briefly about them and demonstrate helpful commands.

To summarize, renaming files is easy and worry-free. Do not be intimidated by a file marked as deleted – a **datalad save** will rectify this. Be mindful of other modifications in your dataset, though, and either supply appropriate paths to `datalad save`, or use Git tools to exclusively save the name change and nothing else.

Let's revert this now, to have a clean history.

```
$ git reset --hard HEAD~1
$ datalad status
HEAD is now at 1fae7c2 add container and execute analysis within container
nothing to save, working tree clean
```

Moving files from or into subdirectories

Let's move an annexed file from within books/ into the root of the superdataset:

```
$ mv TLCL.pdf ../TLCL.pdf
$ datalad status
untracked: /home/me/dl-101/DataLad-101/TLCL.pdf (symlink)
  deleted: /home/me/dl-101/DataLad-101/books/TLCL.pdf (symlink)
```

In general, this looks exactly like renaming or moving a file in the same directory. There is a subtle difference though: Currently, the symlink of the annexed file is broken. There are two ways to demonstrate this. One is trying to open the file – this will currently fail. The second way is to look at the symlink:

```
$ cd ../
$ ls -l TLCL.pdf
lrwxrwxrwx 1 adina adina 131 Jul 29 16:25 TLCL.pdf -> ../.git/annex/objects/jf/3M/MD5E-
↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf
```

The first part of the symlink should point into the .git/ directory, but currently, it does not – the symlink still looks like TLCL.pdf would be within books/. Instead of pointing into .git, it currently points to ../.git, which is non-existent, and even outside of the superdataset. This is

---

why the file cannot be opened: When any program tries to follow the symlink, it will not resolve, and an error such as "no file or directory" will be returned. But do not panic! A `datalad save` will rectify this as well:

```
$ datalad save -m "moved book into root"
$ ls -l TLCL.pdf
delete(ok): books/TLCL.pdf (file)
add(ok): TLCL.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  delete (ok: 1)
  save (ok: 1)
lrwxrwxrwx 1 adina adina 128 Jul 29 16:25 TLCL.pdf -> .git/annex/objects/jf/3M/MD5E-
↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf
```

After a `datalad save`, the symlink is fixed again. Therefore, in general, whenever moving or renaming a file, especially between directories, a `datalad save` is the best option to turn to. Therefore, while it might be startling if you've moved a file and can not open it directly afterwards, everything will be rectified by `datalad save` as well.

> 💡 **M14.2 Why a move between directories is actually a content change**
>
> Let's see how this shows up in the dataset history:
>
> ```
> $ git log -n 1 -p
> commit cdff14955d0175d57af4beee3316b45c54195632
> Author: Elena Piscopia <elena@example.net>
> Date:   Thu Jul 29 16:25:03 2021 +0200
>
>     moved book into root
>
> diff --git a/TLCL.pdf b/TLCL.pdf
> new file mode 120000
> index 0000000..34328e2
> --- /dev/null
> +++ b/TLCL.pdf
> @@ -0,0 +1 @@
> +.git/annex/objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-
> ↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
> \ No newline at end of file
> diff --git a/books/TLCL.pdf b/books/TLCL.pdf
> deleted file mode 120000
> index 4c84b61..0000000
> --- a/books/TLCL.pdf
> +++ /dev/null
> @@ -1 +0,0 @@
> -../.git/annex/objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/
> ↪MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
> \ No newline at end of file
> ```
>
> As you can see, this action does not show up as a move, but instead a deletion and addition of a new file. Why? Because the content that is tracked is the actual symlink, and due to the change in relative location, the symlink needed to change. Hence, what looks and feels like a move on the file system for you is actually a move plus a content

💡 change for Git.

> **G14.1 git annex fix**
>
> A **datalad save** command internally uses a **git commit** to save changes to a dataset. **git commit** in turn triggers a **git annex fix** command. This git-annex command fixes up links that have become broken to again point to annexed content, and is responsible for cleaning up what needs to be cleaned up. Thanks, git-annex!

Finally, let's clean up:

```
$ git reset --hard HEAD~1
HEAD is now at 1fae7c2 add container and execute analysis within container
```

Moving files across dataset boundaries

Generally speaking, moving files across dataset hierarchies is not advised. While DataLad blurs the dataset boundaries to ease working in nested dataset, the dataset boundaries do still exist. If you move a file from one subdataset into another, or up or down a dataset hierarchy, you will move it out of the version control it was in (i.e., from one .git directory into a different one). From the perspective of the first subdataset, the file will be deleted, and from the perspective of the receiving dataset, the file will be added to the dataset, but straight out of nowhere, with none of its potential history from its original dataset attached to it. Before moving a file, consider whether *copying* it (outlined in the next but one paragraph) might be a more suitable alternative.

If you are willing to sacrifice[298] the file's history and move it to a different dataset, the procedure differs between annexed files, and files stored in Git.

For files that Git manages, moving and saving is simple: Move the file, and save the resulting changes in *both* affected datasets (this can be done with a recursive **save** from a top-level dataset, though).

```
$ mv notes.txt midterm_project/notes.txt
$ datalad status -r
 modified: midterm_project (dataset)
untracked: midterm_project/notes.txt (file)
  deleted: notes.txt (file)

$ datalad save -r -m "moved notes.txt from root of top-ds to midterm subds"
add(ok): notes.txt (file)
save(ok): midterm_project (dataset)
delete(ok): notes.txt (file)
add(ok): midterm_project (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  delete (ok: 1)
  save (notneeded: 2, ok: 2)
```

---

[298] Or rather: split – basically, the file is getting a fresh new start. Think of it as some sort of witness-protection program with complete disrespect for provenance. . .

14.2. Miscellaneous file system operations   **229**

Note how the history of notes.txt does not exist in the subdataset – it appears as if the file was generated at once, instead of successively over the course:

```
$ cd midterm_project
$ git log notes.txt
commit 73706bbae3ccd44dd1b8f7250fd711e591a517e7
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:05 2021 +0200

    moved notes.txt from root of top-ds to midterm subds
```

(Undo-ing this requires git resets in *both* datasets)

```
# in midterm_project
$ git reset --hard HEAD~

# in DataLad-101
$ cd ../
$ git reset --hard HEAD~
HEAD is now at e231e46 [DATALAD RUNCMD] rerun analysis in container
HEAD is now at 1fae7c2 add container and execute analysis within container
```

The process is a bit more complex for annexed files. Let's do it wrong, first: What happens if we move an annexed file in the same way as notes.txt?

```
$ mv books/TLCL.pdf midterm_project
$ datalad status -r
  deleted: books/TLCL.pdf (symlink)
 modified: midterm_project (dataset)
untracked: midterm_project/TLCL.pdf (symlink)
```

```
$ datalad save -r -m "move annexed file around"
add(ok): TLCL.pdf (file) [  TLCL.pdf is a git-annex symlink. Its content is not available␣
↪in this repository. (Maybe TLCL.pdf was copied from another repository?)]
save(ok): midterm_project (dataset)
delete(ok): books/TLCL.pdf (file)
add(ok): midterm_project (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  delete (ok: 1)
  save (notneeded: 2, ok: 2)
```

At this point, this does not look that different to the result of moving notes.txt. Note, though, that the deleted and untracked PDFs are symlinks – and therein lies the problem: What was moved was not the file content (which is still in the annex of the top-level dataset, DataLad-101), but its symlink that was stored in Git. After moving the file, the symlink is broken, and git-annex has no way of finding out where the file content could be:

```
$ cd midterm_project
$ git annex whereis TLCL.pdf
whereis TLCL.pdf (0 copies) failed
git-annex: whereis: 1 failed
```

Let's rewind, and find out how to do it correctly:

```
$ git reset --hard HEAD~
$ cd ../
$ git reset --hard HEAD~
HEAD is now at e231e46 [DATALAD RUNCMD] rerun analysis in container
HEAD is now at 1fae7c2 add container and execute analysis within container
```

The crucial step to remember is to get the annexed file out of the annex prior to moving it. For this, we need to fall back to git-annex commands:

```
$ git annex unlock books/TLCL.pdf
$ mv books/TLCL.pdf midterm_project
$ datalad status -r
unlock books/TLCL.pdf ok
(recording state in git...)
  deleted: books/TLCL.pdf (file)
 modified: midterm_project (dataset)
untracked: midterm_project/TLCL.pdf (file)
```

Afterwards, a (recursive) **save** commits the removal of the book from DataLad-101, and adds the file content into the annex of midterm_project:

```
$ datalad save -r -m "move book into midterm_project"
add(ok): TLCL.pdf (file)
save(ok): midterm_project (dataset)
delete(ok): books/TLCL.pdf (file)
add(ok): midterm_project (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  delete (ok: 1)
  save (notneeded: 2, ok: 2)
```

Even though you did split the file's history, at least its content is in the correct dataset now:

```
$ cd midterm_project
$ git annex whereis TLCL.pdf
whereis TLCL.pdf (1 copy)
        eeb3bae2-0e0c-41f3-b759-e6b76fd48f1a -- me@muninn:~/dl-101/DataLad-101/midterm_
↪project [here]
ok
```

But more than showing you how it can be done, if necessary, this paragraph hopefully convinced you that moving files across dataset boundaries is not convenient. It can be a confusing and potentially "file-content-losing"-dangerous process, but it also dissociates a file from its provenance that is captured in its previous dataset, with no machine-readable way to learn about the move easily. A better alternative may be copying files with the **datalad copy-file** command introduced in detail in *Subsample datasets using datalad copy-file* (page 313), and demonstrated in the next but one paragraph. Let's quickly clean up by moving the file back:

```
# in midterm_project
$ git annex unannex TLCL.pdf
unannex TLCL.pdf ok
(recording state in git...)
```

```
$ mv TLCL.pdf ../books
$ cd ../
$ datalad save -r -m "move book back from midterm_project"
save(ok): midterm_project (dataset)
add(ok): midterm_project (file)
add(ok): books/TLCL.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  save (notneeded: 2, ok: 2)
```

Copying files

Let's create a copy of an annexed file, using the Unix command cp to copy.

```
$ cp books/TLCL.pdf copyofTLCL.pdf
$ datalad status
untracked: copyofTLCL.pdf (file)
```

That's expected. The copy shows up as a new, untracked file. Let's save it:

```
$ datalad save -m "add copy of TLCL.pdf"
add(ok): copyofTLCL.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

```
$ git log -n 1 -p
commit 8049f0ace5bc3268e7b9a7b3bc2030b4a3859bfb
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:11 2021 +0200

    add copy of TLCL.pdf

diff --git a/copyofTLCL.pdf b/copyofTLCL.pdf
new file mode 120000
index 0000000..34328e2
--- /dev/null
+++ b/copyofTLCL.pdf
@@ -0,0 +1 @@
+.git/annex/objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-
↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
\ No newline at end of file
```

That's it.

> **M14.3 Symlinks!**
>
> If you have read the additional content in the section *Data integrity* (page 85), you know that the same file content is only stored once, and copies of the same file point to the same location in the object tree.
> Let's check that out:

> 💡
> ```
> $ ls -l copyofTLCL.pdf
> $ ls -l books/TLCL.pdf
> lrwxrwxrwx 1 adina adina 128 Jul 29 16:25 copyofTLCL.pdf -> .git/annex/objects/jf/
> ↪3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
> ↪06d1efcb05bb2c55cd039dab3fb28455.pdf
> lrwxrwxrwx 1 adina adina 131 Jan 28  2019 books/TLCL.pdf -> ../.git/annex/objects/
> ↪jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
> ↪06d1efcb05bb2c55cd039dab3fb28455.pdf
> ```
>
> Indeed! Apart from their relative location (`.git` versus `../.git`) their symlink is identical.
> Thus, even though two copies of the book exist in your dataset, your disk needs to store
> it only once.
>
> In most cases, this is just an interesting fun-fact, but beware when dropping content with
> **datalad drop** (*Removing annexed content entirely* (page 241)): If you drop the content of
> one copy of a file, all other copies will lose this content as well.

Finally, let's clean up:

```
$ git reset --hard HEAD~1
HEAD is now at 324a09b move book back from midterm_project
```

Copying files across dataset boundaries

> ⚠ **copy-file availability**
>
> **datalad copy-file** requires DataLad version `0.13.0` or higher.

Instead of moving files across dataset boundaries, *copying* them is an easier and – **beginning
with DataLad version 0.13.0** – actually supported method. The DataLad command that can
be used for this is **datalad copy-file** (datalad-copy-file manual). This command allows to
copy files (from any dataset or non-dataset location, annexed or not annexed) into a dataset. If
the file is copied from a dataset and is annexed, its availability metadata is added to the new
dataset as well, and there is no need for unannex'ing the or even retrieving its file contents.
Let's see this in action for a file stored in Git, and a file stored in annex:

```
$ datalad copy-file notes.txt midterm_project -d midterm_project
[INFO] Copying non-annexed file or copy into non-annex dataset: /home/me/dl-101/DataLad-
↪101/notes.txt -> AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project)
copy_file(ok): /home/me/dl-101/DataLad-101/notes.txt
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  copy_file (ok: 1)
  save (ok: 1)


$ datalad copy-file books/bash_guide.pdf midterm_project -d midterm_project
copy_file(ok): /home/me/dl-101/DataLad-101/books/bash_guide.pdf [/home/me/dl-101/DataLad-
↪101/midterm_project/bash_guide.pdf]
save(ok): . (dataset)
action summary:
```

(continues on next page)

```
copy_file (ok: 1)
save (ok: 1)
```

Both files have been successfully transferred and saved to the subdataset, and no unannexing was necessary. Note, though, that notes.txt was annexed in the subdataset, as this subdataset was not configured with the text2git RUN PROCEDURE.

```
$ tree midterm_project
midterm_project
├── bash_guide.pdf -> .git/annex/objects/WF/Gq/MD5E-s1198170--
→0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
├── CHANGELOG.md
├── code
│   ├── README.md
│   └── script.py
├── input
│   └── iris.csv -> .git/annex/objects/qz/Jg/MD5E-s3975--341a3b5244f213282b7b0920b729c592.
→csv/MD5E-s3975--341a3b5244f213282b7b0920b729c592.csv
├── notes.txt -> .git/annex/objects/mf/wJ/MD5E-s5074--99d027490a2f9a9c49cffc2c34b55d5c.
→txt/MD5E-s5074--99d027490a2f9a9c49cffc2c34b55d5c.txt
├── pairwise_relationships.png -> .git/annex/objects/q1/gp/MD5E-s261062--
→025dc493ec2da6f9f79eb1ce8512cbec.png/MD5E-s261062--025dc493ec2da6f9f79eb1ce8512cbec.png
├── prediction_report.csv -> .git/annex/objects/8q/6M/MD5E-s345--
→a88cab39b1a5ec59ace322225cc88bc9.csv/MD5E-s345--a88cab39b1a5ec59ace322225cc88bc9.csv
└── README.md

2 directories, 9 files
```

The subdataset has two new commits as **datalad copy-file** can take care of saving changes in the copied-to dataset, and thus the new subdataset state would need to be saved in the superdataset.

```
$ datalad status -r
 modified: midterm_project (dataset)
```

Still, just as when we *moved* files across dataset boundaries, the files' provenance record is lost:

```
$ cd midterm_project
$ git log notes.txt
commit d75fdadbb0e91ef4bcdc2587e02e6dd8b965f0d5
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:12 2021 +0200

    [DATALAD] Recorded changes
```

Nevertheless, copying files with **datalad copy-file** is easier and safer than moving them with standard Unix commands, especially so for annexed files. A more detailed introduction to **datalad copy-file** and a concrete usecase can currently be found in *Subsample datasets using datalad copy-file* (page 313).

Let's clean up:

```
$ git reset --hard HEAD~2
HEAD is now at a3cd3ca move book back from midterm_project
```

Moving/renaming a subdirectory or subdataset

Moving or renaming subdirectories, especially if they are subdatasets, *can* be a minefield. But in principle, a safe way to proceed is using the Unix **mv** command to move or rename, and the **datalad save** to clean up afterwards, just as in the examples above. Make sure to **not** use git mv, especially for subdatasets.

Let's for example rename the books directory:

```
$ mv books/ readings
$ datalad status
untracked: readings (directory)
  deleted: books/TLCL.pdf (symlink)
  deleted: books/bash_guide.pdf (symlink)
  deleted: books/byte-of-python.pdf (symlink)
  deleted: books/progit.pdf (symlink)

$ datalad save -m "renamed directory"
delete(ok): books/TLCL.pdf (file)
delete(ok): books/bash_guide.pdf (file)
delete(ok): books/byte-of-python.pdf (file)
delete(ok): books/progit.pdf (file)
add(ok): readings/TLCL.pdf (file)
add(ok): readings/bash_guide.pdf (file)
add(ok): readings/byte-of-python.pdf (file)
add(ok): readings/progit.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 4)
  delete (ok: 4)
  save (ok: 1)
```

This is easy, and complication free. Moving (as in: changing the location, instead of the name) the directory would work in the same fashion, and a **datalad save** would fix broken symlinks afterwards. Let's quickly clean this up:

```
$ git reset --hard HEAD~1
HEAD is now at 324a09b move book back from midterm_project
```

But let's now try to move the longnow subdataset into the root of the superdataset:

```
$ mv recordings/longnow .
$ datalad status
untracked: longnow (directory)
  deleted: recordings/longnow (dataset)

$ datalad save -m "moved subdataset"
delete(ok): recordings/longnow (file)
add(ok): longnow (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  delete (ok: 1)
  save (ok: 1)
```

```
$ datalad status
nothing to save, working tree clean
```

This seems fine, and it has indeed worked. However, *reverting* a commit like this is tricky, at the moment. This could lead to trouble if you at a later point try to revert or rebase chunks of your history including this move. Therefore, if you can, try not to move subdatasets around. For now we'll clean up in a somewhat "hacky" way: Reverting, and moving remaining subdataset contents back to their original place by hand to take care of the unwanted changes the commit reversal introduced.

```
$ git reset --hard HEAD~1
warning: unable to rmdir 'longnow': Directory not empty
HEAD is now at 324a09b move book back from midterm_project


$ mv -f longnow recordings
```

The take-home message therefore is that it is best not to move subdatasets, but very possible to move subdirectories if necessary. In both cases, do not attempt moving with the **git mv**, but stick with **mv** and a subsequent **datalad save**.

Moving/renaming a superdataset

Once created, a DataLad superdataset may not be in an optimal place on your file system, or have the best name.

After a while, you might think that the dataset would fit much better into /home/user/ research_projects/ than in /home/user/Documents/MyFiles/tmp/datalad-test/. Or maybe at some point, a long name such as My-very-first-DataLad-project-wohoo-I-am-so-excited does not look pretty in your terminal prompt anymore, and going for finance-2019 seems more professional.

These will be situations in which you want to rename or move a superdataset. Will that break anything?

In all standard situations, no, it will be completely fine. You can use standard Unix commands such as mv to do it, and also whichever graphical user interface or explorer you may use.

Beware of one thing though: If your dataset either is a sibling or has a sibling with the source being a path, moving or renaming the dataset will break the linkage between the datasets. This can be fixed easily though. We can try this in the following hidden section.

> 💡 **M14.4 If a renamed/moved dataset is a sibling...**
>
> As section *DIY configurations* (page 112) explains, each sibling is registered in .git/ config in a "submodule" section. Let's look at how our sibling "roommate" is registered there:

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        editor = nano
[annex]
        uuid = 142a8659-5566-466d-af37-5958d071a995
        version = 8
        backends = MD5E
[filter "annex"]
        smudge = git-annex smudge -- %f
        clean = git-annex smudge --clean -- %f
[submodule "recordings/longnow"]
        active = true
        url = https://github.com/datalad-datasets/longnow-podcasts.git
[remote "roommate"]
        url = ../mock_user/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
        annex-uuid = dfa64995-4862-40bf-b5c7-518db36f3f4d
        annex-ignore = false
[submodule "midterm_project"]
        active = true
        url = ./midterm_project
[submodule "longnow"]
        active = true
        url = https://github.com/datalad-datasets/longnow-podcasts.git
```

As you can see, its "url" is specified as a relative path. Say your room mate's directory is
a dataset you would want to move. Let's see what happens if we move the dataset such
that the path does not point to the dataset anymore:

```
# add an intermediate directory
$ cd ../mock_user
$ mkdir onemoredir
# move your room mates dataset into this new directory
$ mv DataLad-101 onemoredir
```

This means that relative to your DataLad-101, your room mates dataset is not at ../
mock_user/DataLad-101 anymore, but in ../mock_user/onemoredir/DataLad-101. The
path specified in the configuration file is thus wrong now.

```
# navigate back into your dataset
$ cd ../DataLad-101
# attempt a datalad update
$ datalad update
[INFO] Fetching updates for Dataset(/home/me/dl-101/DataLad-101)
CommandError: 'git fetch --verbose --progress --prune --recurse-submodules=no␣
↪roommate' failed with exitcode 128 under /home/me/dl-101/DataLad-101
fatal: '../mock_user/DataLad-101' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Here we go:

14.2. Miscellaneous file system operations   **237**

💡

```
'fatal: '../mock_user/DataLad-101' does not appear to be a git repository
 fatal: Could not read from remote repository.
```

Git seems pretty insistent (given the amount of error messages) that it can not seem to find a Git repository at the location the `.git/config` file specified. Luckily, we can provide this information. Edit the file with an editor of your choice and fix the path from `url = ../mock_user/DataLad-101` to `url = ../mock_user/onemoredir/DataLad-101`. Below, we are using the stream editor sed[294] for this operation.

```
$ sed -i 's/..\/mock_user\/DataLad-101/..\/mock_user\/onemoredir\/DataLad-101/' .
↪git/config
```

This is how the file looks now:

```
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        editor = nano
[annex]
        uuid = 142a8659-5566-466d-af37-5958d071a995
        version = 8
        backends = MD5E
[filter "annex"]
        smudge = git-annex smudge -- %f
        clean = git-annex smudge --clean -- %f
[submodule "recordings/longnow"]
        active = true
        url = https://github.com/datalad-datasets/longnow-podcasts.git
[remote "roommate"]
        url = ../mock_user/onemoredir/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
        annex-uuid = dfa64995-4862-40bf-b5c7-518db36f3f4d
        annex-ignore = false
[submodule "midterm_project"]
        active = true
        url = ./midterm_project
[submodule "longnow"]
        active = true
        url = https://github.com/datalad-datasets/longnow-podcasts.git
```

Let's try to update now:

```
$ datalad update
[INFO] Fetching updates for Dataset(/home/me/dl-101/DataLad-101)
update(ok): . (dataset)
```

Nice! We fixed it! Therefore, if a dataset you move or rename is known to other datasets from its path, or identifies siblings with paths, make sure to adjust them in the `.git/config` file.

To clean up, we'll redo the move of the dataset and the modification in `.git/config`.

> 💡
> ```
> $ cd ../mock_user && mv onemoredir/DataLad-101 .
> $ rm -r onemoredir
> $ cd ../DataLad-101 && sed -i 's/..\/mock_user\/onemoredir\/DataLad-101/..\/mock_
> →user\/DataLad-101/' .git/config
> ```
> ─────────────────
> ²⁹⁴ https://en.wikipedia.org/wiki/Sed

Getting contents out of git-annex

Files in your dataset can either be handled by GIT or GIT-ANNEX. Self-made or predefined configurations to .gitattributes, defaults, or the --to-git option to **datalad save** allow you to control which tool does what on up to single-file basis. Accidentally though, you may give a file of yours to git-annex when it was intended to be stored in Git, or you want to get a previously annexed file into Git.

Consider you intend to share the cropped .png images you created from the longnow logos. Would you publish your DataLad-101 dataset so GITHUB or GITLAB, these files would not be available to others, because annexed dataset contents can not be published to these services. Even though you could find a third party service of your choice and publish your dataset *and* the annexed data (see section *Beyond shared infrastructure* (page 177)), you're feeling lazy today. And since it is only two files, and they are quite small, you decide to store them in Git – this way, the files would be available without configuring an external data store.

To get contents out of the dataset's annex you need to *unannex* them. This is done with the git-annex command **git annex unannex**. Let's see how it works:

```
$ git annex unannex recordings/*logo_small.jpg
unannex recordings/interval_logo_small.jpg ok
unannex recordings/salt_logo_small.jpg ok
(recording state in git...)
```

Your dataset's history records the unannexing of the files.

```
$ git log -p -n 1
commit 324a09b6d97b867ee5896b0ec5af25db716d9c3f
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:10 2021 +0200

    move book back from midterm_project

diff --git a/books/TLCL.pdf b/books/TLCL.pdf
new file mode 120000
index 0000000..4c84b61
--- /dev/null
+++ b/books/TLCL.pdf
@@ -0,0 +1 @@
+../.git/annex/objects/jf/3M/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-
→s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
\ No newline at end of file
diff --git a/midterm_project b/midterm_project
index 566e7d3..a3cd3ca 160000
--- a/midterm_project
+++ b/midterm_project
```

(continues on next page)

```
@@ -1 +1 @@
-Subproject commit 566e7d3aab39017a90b1afb68f9cb15317363c18
+Subproject commit a3cd3cad564736d67d90b3a77561c28b05e902ef
```

Once files have been unannexed, they are "untracked" again, and you can save them into Git,
either by adding a rule to `.gitattributes`, or with **datalad save --to-git**:

```
$ datalad save --to-git -m "save cropped logos to Git" recordings/*jpg
add(ok): recordings/interval_logo_small.jpg (file)
add(ok): recordings/salt_logo_small.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  save (ok: 1)
```

Deleting (annexed) files/directories

Removing annexed file content from a dataset is possible in two different ways: Either by
removing the file from the current state of the repository (which Git calls the *worktree*) but
keeping the content in the history of the dataset, or by removing content entirely from a dataset
and its history.

Removing a file, but keeping content in history

An `rm <file>` or `rm -rf <directory>` with a subsequent **datalad save** will remove a file or
directory, and save its removal. The file content however will still be in the history of the dataset,
and the file can be brought back to existence by going back into the history of the dataset or
reverting the removal commit:

```
# download a file
$ datalad download-url -m "Added flower mosaic from wikimedia" \
  https://upload.wikimedia.org/wikipedia/commons/a/a5/Flower_poster_2.jpg \
  --path flowers.jpg
$ ls -l flowers.jpg
[INFO] Downloading 'https://upload.wikimedia.org/wikipedia/commons/a/a5/Flower_poster_2.
↪jpg' into '/home/me/dl-101/DataLad-101/flowers.jpg'
download_url(ok): /home/me/dl-101/DataLad-101/flowers.jpg (file)
add(ok): flowers.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
lrwxrwxrwx 1 adina adina 128 Oct  8  2014 flowers.jpg -> .git/annex/objects/7q/9Z/MD5E-
↪s4487679--3898ef0e3497a89fa1ea74698992bf51.jpg/MD5E-s4487679--
↪3898ef0e3497a89fa1ea74698992bf51.jpg


# removal is easy:
$ rm flowers.jpg
```

This will lead to a dirty dataset status:

```
$ datalad status
  deleted: flowers.jpg (symlink)
```

If a removal happened by accident, a `git checkout -- flowers.jpg` would undo the removal at this stage. To stick with the removal and clean up the dataset state, **datalad save** will suffice:

```
$ datalad save -m "removed file again"
delete(ok): flowers.jpg (file)
save(ok): . (dataset)
action summary:
  delete (ok: 1)
  save (ok: 1)
```

This commits the deletion of the file in the dataset's history. If this commit is reverted, the file comes back to existence:

```
$ git reset --hard HEAD~1
$ ls
HEAD is now at afe8e14 Added flower mosaic from wikimedia
books
code
flowers.jpg
midterm_project
notes.txt
recordings
```

In other words, with an **rm** and subsequent **datalad save**, the symlink is removed, but the content is retained in the history.

Removing annexed content entirely

The command to remove file content entirely and irreversibly from a repository is the **datalad drop** command (datalad-drop manual). This command will delete the content stored in the annex of the dataset, and can be very helpful to make a dataset more lean if the file content is either irrelevant or can be retrieved from other sources easily. Think about a situation in which a very large result file is computed by default in some analysis, but is not relevant for any project, and can thus be removed. Or if only the results of an analysis need to be kept, but the file contents from its input datasets can be dropped at these input datasets are backed-up else where. Because the command works on annexed contents, it will drop file *content* from a dataset, but it will retain the symlink for this file (as this symlink is stored in Git).

drop can take any number of files. If an entire dataset is specified, all file content in sub-*directories* is dropped automatically, but for content in sub-*datasets* to be dropped, the `-r/--recursive` flag has to be included. By default, DataLad will not drop any content that does not have at least one verified remote copy that the content could be retrieved from again. It is possible to drop the downloaded image, because thanks to **datalad download-url** its original location in the web in known:

```
$ datalad drop flowers.jpg
drop(ok): /home/me/dl-101/DataLad-101/flowers.jpg (file) [checking https://upload.
↪wikimedia.org/wikipedia/commons/a/a5/Flower_poster_2.jpg...]
```

Currently, the file content is gone, but the symlink still exist. Opening the remaining symlink will fail, but the content can be obtained easily again with **datalad get**:

```
$ datalad get flowers.jpg
get(ok): flowers.jpg (file) [from web...]
```

If a file has no verified remote copies, DataLad will only drop its content if the --nocheck option is specified. We will demonstrate this by generating a random PDF file:

```
$ convert xc:none -page Letter a.pdf
$ datalad save -m "add empty pdf"
add(ok): a.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

DataLad will safeguard dropping content that it can not retrieve again:

```
$ datalad drop a.pdf
drop(error): /home/me/dl-101/DataLad-101/a.pdf (file) [unsafe; Could only verify the␣
↪existence of 0 out of 1 necessary copies; (Use --nocheck to override this check, or␣
↪adjust numcopies.)]
```

But with the --nocheck flag it will work:

```
$ datalad drop --nocheck a.pdf
drop(ok): /home/me/dl-101/DataLad-101/a.pdf (file)
```

Note though that this file content is irreversibly gone now, and even going back in time in the history of the dataset will not bring it back into existence.

Finally, let's clean up:

```
$ git reset --hard HEAD~2
HEAD is now at 9d34329 save cropped logos to Git
```

Deleting content stored in Git

It is much harder to delete dataset content that is stored in Git compared to content stored in git-annex. Operations such as rm or git rm remove the file from the *worktree*, but not from its history, and they can be brought back to life just as annexed contents that were solely rm'ed. There is also no straightforward Git equivalent of drop. To accomplish a complete removal of a file from a dataset, we recommend the external tool git-filter-repo[295]. It is a powerful and potentially very dangerous tool to rewrite Git history.

Usually, removing files stored in Git completely is not a common or recommended operation, as it involves quite aggressive rewriting of the dataset history. Sometimes, however, sensitive files, for example private SSH KEYs or passwords, or too many or too large files are accidentally saved into Git, and *need* to get out of the dataset history. The command git-filter-repo <path-specification> --force will "filter-out", i.e., remove all files **but the ones specified** in <path-specification> from the datasets history. The section *Fixing up too-large datasets* (page 328) shows an example invocation. If you want to use it, however, make sure to attempt it in a dataset clone or with its --dry-run flag first. It is easy to loose dataset history and files with this tool.

---

[295] https://github.com/newren/git-filter-repo

Chapter 14. Help yourself

Uninstalling or deleting subdatasets

Depending on the exact aim, two commands are of relevance for deleting a DataLad subdataset. The softer (and not so much "deleting" version) is to uninstall a dataset with the **datalad uninstall** (datalad-uninstall manual). This command can be used to uninstall any number of *subdatasets*. Note though that only subdatasets can be uninstalled; the command will error if given a sub-*directory*, a file, or a top-level dataset.

```
# clone a subdataset - the content is irrelevant, so why not a cloud :)
$ datalad clone -d . \
 https://github.com/datalad-datasets/disneyanimation-cloud.git \
 cloud
[INFO] Cloning dataset to Dataset(/home/me/dl-101/DataLad-101/cloud)
[INFO] Attempting to clone from https://github.com/datalad-datasets/disneyanimation-cloud.
↪git to /home/me/dl-101/DataLad-101/cloud
[INFO] Start enumerating objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/DataLad-101/cloud)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Remote origin not usable by git-annex; setting annex-ignore
install(ok): cloud (dataset)
add(ok): cloud (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

To uninstall the dataset, use

```
$ datalad uninstall cloud
uninstall(ok): cloud (dataset)
action summary:
  drop (notneeded: 1)
  uninstall (ok: 1)
```

Note that the dataset is still known in the dataset, and not completely removed. A datalad get [-n/--no-data] cloud would install the dataset again.

In case one wants to fully delete a subdataset from a dataset, the **datalad remove** command (datalad-remove manual) is relevant[299]. It needs a pointer to the root of the superdataset with the -d/--dataset flag, a path to the subdataset to be removed, and optionally a commit message (-m/--message) or recursive specification (-r/--recursive). To remove a subdataset, we will install the uninstalled subdataset again, and subsequently remove it with the **datalad remove** command:

```
$ datalad get -n cloud
[INFO] Cloning dataset to Dataset(/home/me/dl-101/DataLad-101/cloud)
[INFO] Attempting to clone from https://github.com/datalad-datasets/disneyanimation-cloud.
↪git to /home/me/dl-101/DataLad-101/cloud
[INFO] Start enumerating objects
```

(continues on next page)

---

[299] This is indeed the only case in which **datalad remove** is relevant. For all other cases of content deletion a normal rm with a subsequent **datalad save** works best.

```
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/DataLad-101/cloud)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Remote origin not usable by git-annex; setting annex-ignore
install(ok): /home/me/dl-101/DataLad-101/cloud (dataset) [Installed subdataset in order␣
↪to get /home/me/dl-101/DataLad-101/cloud]


# delete the subdataset
$ datalad remove -m "remove obsolete subds" -d . cloud
uninstall(ok): cloud (dataset)
remove(ok): cloud (dataset)
save(ok): . (dataset)
action summary:
  drop (notneeded: 1)
  remove (ok: 1)
  save (ok: 1)
  uninstall (ok: 1)
```

Note that for both commands a pointer to the *current directory* will not work. `datalad remove
.` or `datalad uninstall .` will fail, even if the command is executed in a subdataset instead of
the top-level superdataset – you need to execute the command from a higher-level directory.

Deleting a superdataset

If for whatever reason you at one point tried to remove a DataLad dataset, whether with a GUI
or the command line call `rm -rf <directory>`, you likely have seen permission denied errors
such as

```
rm: cannot remove '<directory>/.git/annex/objects/Mz/M1/MD5E-s422982--
↪2977b5c6ea32de1f98689bc42613aac7.jpg/MD5E-s422982--2977b5c6ea32de1f98689bc42613aac7.jpg
↪': Permission denied
rm: cannot remove '<directory>/.git/annex/objects/FP/wv/MD5E-s543180--
↪6209797211280fc0a95196b0f781311e.jpg/MD5E-s543180--6209797211280fc0a95196b0f781311e.jpg
↪': Permission denied
[...]
```

This error indicates that there is write-protected content within `.git` that cannot not be deleted.
What is this write-protected content? It's the file content stored in the object tree of git-annex.
If you want, you can re-read the section on *Data integrity* (page 85) to find out how git-annex
revokes write permission for the user to protect the file content given to it. To remove a dataset
with annexed content one has to regain write permissions to everything in the dataset. This is
done with the chmod[296] command:

```
chmod -R u+w <dataset>
```

This *recursively* (-R, i.e., throughout all files and (sub)directories) gives users (u) write permis-
sions (+w) for the dataset.

Afterwards, `rm -rf <dataset>` will succeed.

---
[296] https://en.wikipedia.org/wiki/Chmod

However, instead of `rm -rf`, a faster way to remove a dataset is using **datalad remove**: Run `datalad remove <dataset>` outside of the superdataset to remove a top-level dataset with all its contents. Likely, both `--nocheck` and `--recursive` flags are necessary to remove content that does not have verified remotes, and to traverse into subdatasets.

Be aware though that both ways to delete a dataset will irretrievably delete the dataset, it's contents, and it's history.

Summary

To sum up, file system management operations are safe and easy. Even if you are currently confused about one or two operations, worry not – the take-home-message is simple: Use `datalad save` whenever you move or rename files. Be mindful that a `datalad status` can appear unintuitive or that symlinks can break if annexed files are moved, but all of these problems are solved after a **datalad save** command. Apart from this command, having a clean dataset status prior to doing anything is your friend as well. It will make sure that you have a neat and organized commit history, and no accidental commits of changes unrelated to your file system management operations. The only operation you should beware of is moving subdatasets around – this can be a minefield. With all of these experiences and tips, you feel confident that you know how to handle your datasets files and directories well and worry-free.

14.3 Back and forth in time

Almost everyone inadvertently deleted or overwrote files at some point with a hasty operation that caused data fatalities or at least troubles to re-obtain or restore data. With DataLad, no mistakes are forever: One powerful feature of datasets is the ability to revert data to a previous state and thus view earlier content or correct mistakes. As long as the content was version controlled (i.e., tracked), it is possible to look at previous states of the data, or revert changes – even years after they happened – thanks to the underlying version control system GIT.
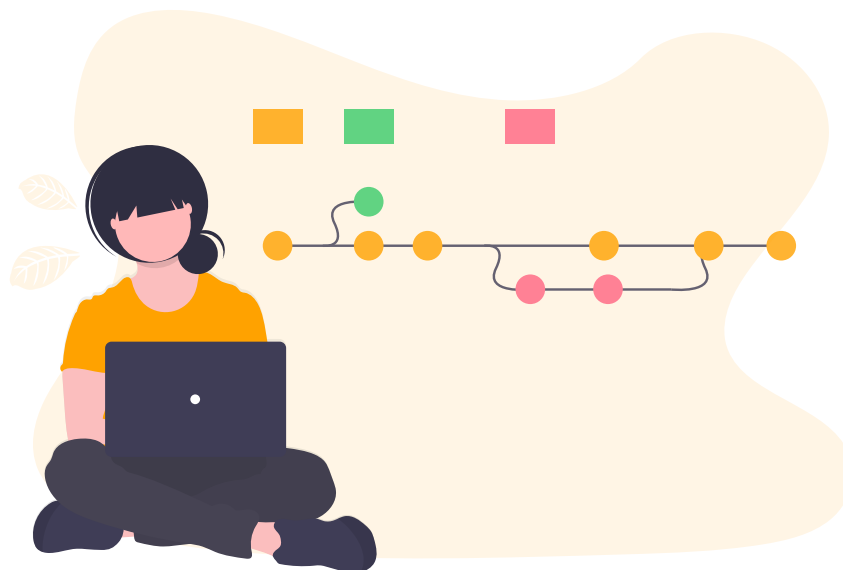
To get a glimpse into how to work with the history of a dataset, today's lecture has an external Git-expert as a guest lecturer. "I do not have enough time to go through all the details in only one lecture. But I'll give you the basics, and an idea of what is possible. Always remember: Just google what you need. You will find thousands of helpful tutorials or questions on Stack

Overflow[300] right away. Even experts will *constantly* seek help to find out which Git command to use, and how to use it.", he reassures with a wink.

The basis of working with the history is to *look at it* with tools such as TIG, GITK, or simply the `git log` command. The most important information in an entry (commit) in the history is the SHASUM (or hash) associated with it. This hash is how dataset modifications in the history are identified, and with this hash you can communicate with DataLad or GIT about these modifications or version states[304]. Here is an excerpt from the DataLad-101 history to show a few abbreviated hashes of the 15 most recent commits[305]:

```
$ git log -15 --oneline
8dc1281 remove obsolete subds
64abc8d [DATALAD] modified subdataset properties
072fbda [DATALAD] Recorded changes
9d34329 save cropped logos to Git
324a09b move book back from midterm_project
ce929ad move book into midterm_project
1fae7c2 add container and execute analysis within container
4d216b8 finished my midterm project
35ca35c [DATALAD] Recorded changes
2d253b4 add note on DataLad's procedures
cf02939 add note on configurations and git config
d58f5ba Add note on adding siblings
67a7a5e Merge remote-tracking branch 'roommate/master'
e75e2df Include nesting demo from datalad website
4069dac add note about datalad update
```

"I'll let you people direct this lecture", the guest lecturer proposes. "You tell me what you would be interested in doing, and I'll show you how it's done. For the rest of the lecture, call me Google!"

Fixing (empty) commit messages

From the back of the lecture hall comes a question you're really glad someone asked: "It has happened to me that I accidentally did a `datalad save` and forgot to specify the commit message, how can I fix this?". The room nods in agreement – apparently, others have run into this premature slip of the Enter key as well.

Let's demonstrate a simple example. First, let's create some random files. Do this right in your dataset.

```
$ cat << EOT > Gitjoke1.txt
Git knows what you did last summer!
EOT
```

```
$ cat << EOT > Gitjoke2.txt
Knock knock. Who's there? Git.
Git-who?
```

(continues on next page)

---

[300] https://stackoverflow.com

[304] For example, the `datalad rerun` command introduced in section *DataLad, Re-Run!* (page 64) takes such a hash as an argument, and re-executes the datalad run or datalad rerun RUN RECORD associated with this hash. Likewise, the `git diff` can work with commit hashes.

[305] There are other alternatives to reference commits in the history of a dataset, for example "counting" ancestors of the most recent commit using the notation HEAD~2, HEAD^2 or HEAD@{2}. However, using hashes to reference commits is a very fail-save method and saves you from accidentally miscounting.

```
Sorry, 'who' is not a git command - did you mean 'show'?
EOT

$ cat << EOT > Gitjoke3.txt
In Soviet Russia, git commits YOU!
EOT
```

This will generate three new files in your dataset. Run a **datalad status** to verify this:

```
$ datalad status
untracked: Gitjoke1.txt (file)
untracked: Gitjoke2.txt (file)
untracked: Gitjoke3.txt (file)
```

And now:

```
$ datalad save
add(ok): Gitjoke1.txt (file)
add(ok): Gitjoke2.txt (file)
add(ok): Gitjoke3.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  save (ok: 1)
```

Whooops! A **datalad save** without a commit message that saved all of the files.

```
$ git log -p -1
commit 4be0dfb8cdc258975dbf1481a94c8e42e3802197
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:37 2021 +0200

    [DATALAD] Recorded changes

diff --git a/Gitjoke1.txt b/Gitjoke1.txt
new file mode 100644
index 0000000..d7e1359
--- /dev/null
+++ b/Gitjoke1.txt
@@ -0,0 +1 @@
+Git knows what you did last summer!
diff --git a/Gitjoke2.txt b/Gitjoke2.txt
new file mode 100644
index 0000000..51beecb
--- /dev/null
+++ b/Gitjoke2.txt
@@ -0,0 +1,3 @@
+Knock knock. Who's there? Git.
+Git-who?
+Sorry, 'who' is not a git command - did you mean 'show'?
diff --git a/Gitjoke3.txt b/Gitjoke3.txt
new file mode 100644
index 0000000..7b83d95
--- /dev/null
+++ b/Gitjoke3.txt
@@ -0,0 +1 @@
```

```
+In Soviet Russia, git commits YOU!
```

As expected, all of the modifications present prior to the command are saved into the most recent commit, and the commit message DataLad provides by default, `[DATALAD] Recorded changes`, is not very helpful.

Changing the commit message of the most recent commit can be done with the command `git commit --amend`. Running this command will open an editor (the default, as configured in Git), and allow you to change the commit message. Make sure to read the *find-out-more on changing other than the most recent commit* (page 278) in case you want to improve the commit message of more commits than only the latest.

Try running the `git commit --amend` command right now and give the commit a new commit message (you can just delete the one created by DataLad in the editor)!

Untracking accidentally saved contents (tracked in Git)

The next question comes from the front: "It happened that I forgot to give a path to the `datalad save` command when I wanted to only start tracking a very specific file. Other times I just didn't remember that additional, untracked files existed in the dataset and saved unaware of those. I know that it is good practice to only save those changes together that belong together, so is there a way to disentangle an accidental `datalad save` again?"

Let's say instead of saving *all three* previously untracked Git jokes you intended to save *only one* of those files. What we want to achieve is to keep all of the files and their contents in the dataset, but get them out of the history into an *untracked* state again, and save them *individually* afterwards.

> ⚠️ **Untracking is different for Git versus git-annex!**
>
> Note that this is a case with *text files* (stored in Git)! For accidental annexing of files, please make sure to check out the next paragraph!

This is a task for the `git reset` command. It essentially allows to undo commits by resetting the history of a dataset to an earlier version. `git reset` comes with several *modes* that determine the exact behavior it, but the relevant one for this aim is `--mixed`[306]. Specifying the command:

```
git reset --mixed COMMIT
```

will preserve all changes made to files since the specified commit in the dataset but remove them from the dataset's history. This means all commits *since* COMMIT (but *not including* COMMIT) will not be in your history anymore and become "untracked files" or "unsaved changes" instead. In other words, the modifications you made in these commits that are "undone" will still be present in your dataset – just not written to the history anymore. Let's try this to get a feel for it.

The COMMIT in the command can either be a hash or a reference with the HEAD pointer.

---

[306] The option `--mixed` is the default mode for a `git reset` command, omitting it (i.e., running just `git reset`) leads to the same behavior. It is explicitly stated in this book to make the mode clear, though.

M14.6 Git terminology: branches and HEADs?

A Git repository (and thus any DataLad dataset) is built up as a tree of commits. A *branch* is a named pointer (reference) to a commit, and allows you to isolate developments. The default branch is called master. HEAD is a pointer to the branch you are currently on, and thus to the last commit in the given branch.
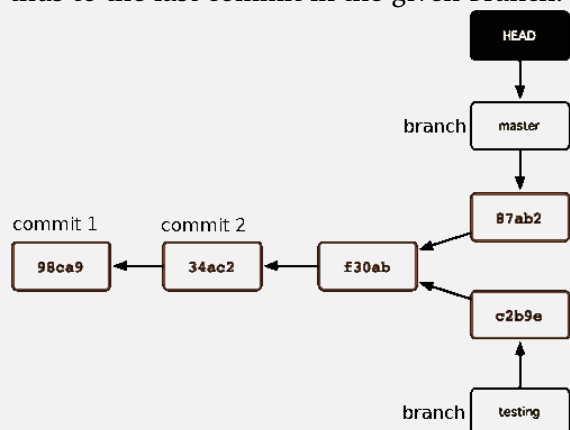
Using HEAD, you can identify the most recent commit, or count backwards starting from the most recent commit. HEAD~1 is the ancestor of the most recent commit, i.e., one commit back (f30ab in the figure above). Apart from the notation HEAD~N, there is also HEAD^N used to count backwards, but less frequently used and of importance primarily in the case of *merge* commits. This post[301] explains the details well.

---

[301] https://stackoverflow.com/questions/2221658/whats-the-difference-between-head-and-head-in-git

Let's stay with the hash, and reset to the commit prior to saving the Gitjokes.

First, find out the shasum, and afterwards, reset it.

```
$ git log -n 3 --oneline
4be0dfb [DATALAD] Recorded changes
8dc1281 remove obsolete subds
64abc8d [DATALAD] modified subdataset properties
```

```
$ git reset --mixed 8dc1281daa00a661e1c6387632b638e945e2f6a6
```

Let's see what has happened. First, let's check the history:

```
$ git log -n 2 --oneline
8dc1281 remove obsolete subds
64abc8d [DATALAD] modified subdataset properties
```

As you can see, the commit in which the jokes were tracked is not in the history anymore! Go on to see what **datalad status** reports:

```
$ datalad status
untracked: Gitjoke1.txt (file)
untracked: Gitjoke2.txt (file)
untracked: Gitjoke3.txt (file)
```

Nice, the files are present, and untracked again. Do they contain the content still? We will read all of them with **cat**:

```
$ cat Gitjoke*
Git knows what you did last summer!
Knock knock. Who's there? Git.
Git-who?
Sorry, 'who' is not a git command - did you mean 'show'?
In Soviet Russia, git commits YOU!
```

Great. Now we can go ahead and save only the file we intended to track:

```
$ datalad save -m "save my favorite Git joke" Gitjoke2.txt
add(ok): Gitjoke2.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Finally, let's check how the history looks afterwards:

```
$ git log -2
commit 81609fc44c66a0f3bcec4c7a5910181dc8c08d6c
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:38 2021 +0200

    save my favorite Git joke

commit 8dc1281daa00a661e1c6387632b638e945e2f6a6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:35 2021 +0200

    remove obsolete subds
```

Wow! You have rewritten history[307] !

Untracking accidentally saved contents (stored in git-annex)

The previous `git reset` undid the tracking of *text* files. However, those files are stored in Git, and thus their content is also stored in Git. Files that are annexed, however, have their content stored in git-annex, and not the file itself is stored in the history, but a symlink pointing to the location of the file content in the dataset's annex. This has consequences for a `git reset` command: Reverting a save of a file that is annexed would revert the save of the symlink into Git, but it will not revert the *annexing* of the file. Thus, what will be left in the dataset is an untracked symlink.

To undo an accidental save of that annexed a file, the annexed file has to be "unlocked" first with a `datalad unlock` command.

We will simulate such a situation by creating a PDF file that gets annexed with an accidental `datalad save`:

```
# create an empty pdf file
$ convert xc:none -page Letter apdffile.pdf
# accidentally save it
$ datalad save
add(ok): Gitjoke1.txt (file)
add(ok): Gitjoke3.txt (file)
```

(continues on next page)

```
add(ok): apdffile.pdf (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  save (ok: 1)
```

This accidental **save** has thus added both text files stored in Git, but also a PDF file to the history of the dataset. As an `ls -l` reveals, the PDF file has been annexed and is thus a SYMLINK:

```
$ ls -l apdffile.pdf
lrwxrwxrwx 1 adina adina 122 Jul 29 16:25 apdffile.pdf -> .git/annex/objects/QZ/Zz/MD5E-
↪s1858--bbcdd91170cf142948f5fb348b060626.pdf/MD5E-s1858--
↪bbcdd91170cf142948f5fb348b060626.pdf
```

Prior to resetting, the PDF file has to be unannexed. To unannex files, i.e., get the contents out of the object tree, the **datalad unlock** command is relevant:

```
$ datalad unlock apdffile.pdf
unlock(ok): apdffile.pdf (file)
```

The file is now no longer symlinked:

```
$ ls -l apdffile.pdf
-rw-r--r-- 1 adina adina 1858 Jul 29 16:25 apdffile.pdf
```

Finally, **git reset --mixed** can be used to revert the accidental **save**. Again, find out the shasum first, and afterwards, reset it.

```
$ git log -n 3 --oneline
8c792b6 [DATALAD] Recorded changes
81609fc save my favorite Git joke
8dc1281 remove obsolete subds
```

```
$ git reset --mixed 81609fc44c66a0f3bcec4c7a5910181dc8c08d6c
```

To see what has happened, let's check the history:

```
$ git log -n 2 --oneline
81609fc save my favorite Git joke
8dc1281 remove obsolete subds
```

. . . and also the status of the dataset:

```
$ datalad status
untracked: Gitjoke1.txt (file)
untracked: Gitjoke3.txt (file)
untracked: apdffile.pdf (file)
```

The accidental save has been undone, and the file is present as untracked content again. As before, this action has not been recorded in your history.

Viewing previous versions of files and datasets

The next question is truly magical: How does one *see* data as it was at a previous state in history?

This magic trick can be performed with the `git checkout`. It is a very heavily used command for various tasks, but among many it can send you back in time to view the state of a dataset at the time of a specific commit.

Let's say you want to find out which notes you took in the first few chapters of the handbook. Find a commit SHASUM in your history to specify the point in time you want to go back to:

```
$ git log -n 20 --oneline
81609fc save my favorite Git joke
8dc1281 remove obsolete subds
64abc8d [DATALAD] modified subdataset properties
072fbda [DATALAD] Recorded changes
9d34329 save cropped logos to Git
324a09b move book back from midterm_project
ce929ad move book into midterm_project
1fae7c2 add container and execute analysis within container
4d216b8 finished my midterm project
35ca35c [DATALAD] Recorded changes
2d253b4 add note on DataLad's procedures
cf02939 add note on configurations and git config
d58f5ba Add note on adding siblings
67a7a5e Merge remote-tracking branch 'roommate/master'
e75e2df Include nesting demo from datalad website
4069dac add note about datalad update
401df53 add note on git annex whereis
dfe8e39 add note about cloning from paths and recursive datalad get
41ea4e6 add note on clean datasets
4753f3f [DATALAD RUNCMD] Resize logo for slides
```

Let's go 15 commits back in time:

```
$ git checkout 401df53b3e9a7a4523e51a5faa33e2572a6cdb2c
warning: unable to rmdir 'midterm_project': Directory not empty
Note: switching to '401df53b3e9a7a4523e51a5faa33e2572a6cdb2c'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 401df53 add note on git annex whereis
```

How did your `notes.txt` file look at this point?

```
$ tail notes.txt

Note that a recursive "datalad get" would install all further
registered subdatasets underneath a subdataset, so a safer way to
proceed is to set a decent --recursion-limit:
"datalad get -n -r --recursion-limit 2 <subds>"

The command "git annex whereis PATH" lists the repositories that have
the file content of an annexed file. When using "datalad get" to
retrieve file content, those repositories will be queried.
```

Neat, isn't it? By checking out a commit shasum you can explore a previous state of a datasets history. And this does not only apply to simple text files, but every type of file in your dataset, regardless of size. The checkout command however led to something that Git calls a "detached HEAD state". While this sounds scary, a **git checkout master** will bring you back into the most recent version of your dataset and get you out of the "detached HEAD state":

```
$ git checkout master
Previous HEAD position was 401df53 add note on git annex whereis
Switched to branch 'master'
```

Note one very important thing: The previously untracked files are still there.

```
$ datalad status
untracked: Gitjoke1.txt (file)
untracked: Gitjoke3.txt (file)
untracked: apdffile.pdf (file)
```

The contents of notes.txt will now be the most recent version again:

```
$ tail notes.txt
configurations, create files or file hierarchies, or perform arbitrary
tasks in datasets. They can be shipped with DataLad, its extensions,
or datasets, and you can even write your own procedures and distribute
them.
The "datalad run-procedure" command is used to apply such a procedure
to a dataset. Procedures shipped with DataLad or its extensions
starting with a "cfg" prefix can also be applied at the creation of a
dataset with "datalad create -c <PROC-NAME> <PATH>" (omitting the
"cfg" prefix).
```

... Wow! You traveled back and forth in time! But an even more magical way to see the contents of files in previous versions is Git's **cat-file** command: Among many other things, it lets you read a file's contents as of any point in time in the history, without a prior **git checkout** (note that the output is shortened for brevity and shows only the last few lines of the file):

```
Note that subdatasets will not be installed by default, but are only
registered in the superdataset -- you will have to do a
"datalad get -n PATH/TO/SUBDATASET" to install the subdataset for file
availability meta data. The -n/--no-data options prevents that file
contents are also downloaded.

Note that a recursive "datalad get" would install all further
registered subdatasets underneath a subdataset, so a safer way to
proceed is to set a decent --recursion-limit:
"datalad get -n -r --recursion-limit 2 <subds>"
```

(continues on next page)

```
The command "git annex whereis PATH" lists the repositories that have
the file content of an annexed file. When using "datalad get" to
retrieve file content, those repositories will be queried.
```

The cat-file command is very versatile, and it's documentation[302] will list all of its functionality. To use it to see the contents of a file at a previous state as done above, this is how the general structure looks like:

```
$ git cat-file --textconv SHASUM:<path/to/file>
```

Undoing latest modifications of files

Previously, we saw how to remove files from a datasets history that were accidentally saved and thus tracked for the first time. How does one undo a *modification* to a tracked file?

Let's modify the saved `Gitjoke1.txt`:

```
$ echo "this is by far my favorite joke!" >> Gitjoke2.txt
```

```
$ cat Gitjoke2.txt
Knock knock. Who's there? Git.
Git-who?
Sorry, 'who' is not a git command - did you mean 'show'?
this is by far my favorite joke!
```

```
$ datalad status
untracked: Gitjoke1.txt (file)
untracked: Gitjoke3.txt (file)
untracked: apdffile.pdf (file)
 modified: Gitjoke2.txt (file)
```

```
$ datalad save -m "add joke evaluation to joke" Gitjoke2.txt
add(ok): Gitjoke2.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

How could this modification to `Gitjoke2.txt` be undone? With the **git reset** command again. If you want to "unsave" the modification but keep it in the file, use **git reset --mixed** as before. However, if you want to get rid of the modifications entirely, use the option `--hard` instead of `--mixed`:

```
$ git log -n 2 --oneline
2fe94e2 add joke evaluation to joke
81609fc save my favorite Git joke
```

```
$ git reset --hard 81609fc44c66a0f3bcec4c7a5910181dc8c08d6c
HEAD is now at 81609fc save my favorite Git joke
```

---

[302] https://git-scm.com/docs/git-cat-file

```
$ cat Gitjoke2.txt
Knock knock. Who's there? Git.
Git-who?
Sorry, 'who' is not a git command - did you mean 'show'?
```

The change has been undone completely. This method will work with files stored in Git and annexed files.

Note that this operation only restores this one file, because the commit that was undone only contained modifications to this one file. This is a demonstration of one of the reasons why one should strive for commits to represent meaningful logical units of change – if necessary, they can be undone easily.

Undoing past modifications of files

What `git reset` did was to undo commits from the most recent version of your dataset. How would one undo a change that happened a while ago, though, with important changes being added afterwards that you want to keep?

Let's save a bad modification to `Gitjoke2.txt`, but also a modification to `notes.txt`:

```
$ echo "bad modification" >> Gitjoke2.txt
```

```
$ datalad save -m "did a bad modification" Gitjoke2.txt
add(ok): Gitjoke2.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

```
$ cat << EOT >> notes.txt

Git has many handy tools to go back in forth in time and work with the
history of datasets. Among many other things you can rewrite commit
messages, undo changes, or look at previous versions of datasets.
A superb resource to find out more about this and practice such Git
operations is this chapter in the Pro-git book:
https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History
EOT
```

```
$ datalad save -m "add note on helpful git resource" notes.txt
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

The objective is to remove the first, "bad" modification, but keep the more recent modification of `notes.txt`. A `git reset` command is not convenient, because resetting would need to reset the most recent, "good" modification as well.

One way to accomplish it is with an *interactive rebase*, using the `git rebase -i` command[308]. Experienced Git-users will know under which situations and how to perform such an interactive rebase.

However, outlining an interactive rebase here in the handbook could lead to problems for readers without (much) Git experience: An interactive rebase, even if performed successfully, can lead to many problems if it is applied with too little experience, for example in any collaborative real-world project.

Instead, we demonstrate a different, less intrusive way to revert one or more changes at any point in the history of a dataset: the **git revert** command. Instead of *rewriting* the history, it will add an additional commit in which the changes of an unwanted commit are reverted.

The command looks like this:

```
$ git revert SHASUM
```

where SHASUM specifies the commit hash of the modification that should be reverted.

> 💡 **M14.7 Reverting more than a single commit**
>
> You can also specify a range of commits like this:
>
> ```
> $ git revert OLDER_SHASUM..NEWERSHASUM
> ```
>
> This command will revert all commits starting with the one after OLDER_SHASUM (i.e. **not including** this commit) until and **including** the one specified with NEWERSHASUM. For each reverted commit, one new commit will be added to the history that reverts it. Thus, if you revert a range of three commits, there will be three reversal commits. If you however want the reversal of a range of commits saved in a single commit, supply the --no-commit option as in
>
> ```
> $ git revert --no-commit OLDER_SHASUM..NEWERSHASUM
> ```
>
> After running this command, run a single git commit to conclude the reversal and save it in a single commit.

Let's see how it looks like:

```
$ git revert f865b2562abb60a942425fbab557e1c25dce877a
[master b1c8ba0] Revert "did a bad modification"
 Date: Thu Jul 29 16:25:44 2021 +0200
 1 file changed, 1 deletion(-)
```

This is the state of the file in which we reverted a modification:

```
$ cat Gitjoke2.txt
Knock knock. Who's there? Git.
Git-who?
Sorry, 'who' is not a git command - did you mean 'show'?
```

It does not contain the bad modification anymore. And this is what happened in the history of the dataset:

```
$ git log -n 3
commit b1c8ba0aa730d7227243fb91ef928d6c14afbbcc
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:44 2021 +0200

    Revert "did a bad modification"
```

(continues on next page)

```
    This reverts commit f865b2562abb60a942425fbab557e1c25dce877a.
```

```
commit b041c7117dd0b8559ba9402a8901fc1a495067de
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:44 2021 +0200

    add note on helpful git resource

commit f865b2562abb60a942425fbab557e1c25dce877a
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 16:25:43 2021 +0200
```

```
    did a bad modification
```

The commit that introduced the bad modification is still present, but it transparently gets undone with the most recent commit. At the same time, the good modification of notes.txt was not influenced in any way. The **git revert** command is thus a transparent and safe way of undoing past changes. Note though that this command can only be used efficiently if the commits in your datasets history are meaningful, independent units – having several unrelated modifications in a single commit may make an easy solution with **git revert** impossible and instead require a complex **checkout**, **revert**, or **rebase** operation.

Finally, let's take a look at the state of the dataset after this operation:

```
$ datalad status
untracked: Gitjoke1.txt (file)
untracked: Gitjoke3.txt (file)
untracked: apdffile.pdf (file)
```

As you can see, unsurprisingly, the **git revert** command had no effects on anything else but the specified commit, and previously untracked files are still present.

Oh no! I'm in a merge conflict!

When working with the history of a dataset, especially when rewriting the history with an interactive rebase or when reverting commits, it is possible to run into so-called *merge conflicts*. Merge conflicts happen when Git needs assistance in deciding which changes to keep and which to apply. It will require you to edit the file the merge conflict is happening in with a text editor, but such merge conflict are by far not as scary as they may seem during the first few times of solving merge conflicts.

This section is not a guide on how to solve merge-conflicts, but a broad overview on the necessary steps, and a pointer to a more comprehensive guide.

- The first thing to do if you end up in a merge conflict is to read the instructions Git is giving you – they are a useful guide.

- Also, it is reassuring to remember that you can always get out of a merge conflict by aborting the operation that led to it (e.g., git rebase --abort).

- To actually solve a merge conflict, you will have to edit files: In the documents the merge conflict applies to, Git marks the sections it needs help with with markers that consists of >, <, and = signs and commit shasums or branch names. There will be two marked parts, and you have to delete the one you do not want to keep, as well as all markers.

- Afterwards, run `git add <path/to/file>` and finally a `git commit`.

An excellent resource on how to deal with merge conflicts is this post[303].

Summary

This guest lecture has given you a glimpse into how to work with the history of your DataLad datasets. To conclude this section, let's remove all untracked contents from the dataset. This can be done with **`git clean`**: The command **`git clean -f`** swipes your dataset clean and removes any untracked file. **Careful! This is not revertible, and content lost with this commands can not be recovered!** If you want to be extra sure, run **`git clean -fn`** beforehand – this will give you a list of the files that would be deleted.

```
$ git clean -f
Removing Gitjoke1.txt
Removing Gitjoke3.txt
Removing apdffile.pdf
```

Afterwards, the **`datalad status`** returns nothing, indicating a clean dataset state with no untracked files or modifications.

```
$ datalad status
nothing to save, working tree clean
```

Finally, if you want, apply your new knowledge about reverting commits to remove the `Gitjoke2.txt` file.

14.4 How to get help

All DataLad errors or problems you encounter during `DataLad-101` are intentional and serve illustrative purposes. But what if you run into any DataLad errors outside of this course? Fortunately, the syllabus has a whole section on that, and on one lazy, warm summer-afternoon you flip through it.

---

[303] https://docs.github.com/en/github/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line

You realize that you already know the most important things: The number one advice on how to get help is "Read the error message."[310]. The second advice it "I'm not kidding: Read the error message[311]. The third advice, finally, says "Honestly, read the f***ing error message[312].

Help yourself

If you run into a DataLad problem and you have followed the three steps above, but the error message does not give you a clue on how to proceed[313], the first you should do is

1. find out which *version* of DataLad you use

2. read the *help page* of the command that failed

The first step is important in order to find out whether a command failed due to using a wrong DataLad version. In order to use this book and follow along, your DataLad version should be datalad-0.12 or higher, for example.

To find out which version you are using, run

```
$ datalad --version
datalad 0.14.6+27.gec839bcf5
```

If you want a comprehensive overview of your full setup, **datalad wtf**[336] is the command to turn to (datalad-wtf manual). Running this command will generate a report about the DataLad installation and configuration. The output below shows an excerpt.

```
$ datalad wtf
# WTF
## configuration <SENSITIVE, report disabled by configuration>
## credentials
  - keyring:
    - active_backends:
      - PlaintextKeyring with no encyption v.1.0 at /home/me/.local/share/python_keyring/
↪keyring_pass.cfg
    - config_file: /home/me/.config/python_keyring/keyringrc.cfg
    - data_root: /home/me/.local/share/python_keyring
## datalad
  - full_version: 0.14.6+27.gec839bcf5
  - version: 0.14.6+27.gec839bcf5
## dataset
  - id: 8e04afb0-af85-4070-be29-858d30d85017
  - metadata: <SENSITIVE, report disabled by configuration>
  - path: /home/me/dl-101/DataLad-101
  - repo: AnnexRepo
## dependencies
  - annexremote: 1.4.5
  - appdirs: 1.4.4
```

---

[310] https://poster.keepcalmandposters.com/default/5986752_keep_calm_and_read_the_error_message.png
[311] https://images.app.goo.gl/GWQ82AAJnx1dWtWx6
[312] https://images.app.goo.gl/ddxg4aowbji6XTrw7
[313] https://imgs.xkcd.com/comics/code_quality_3.png
[336] wtf in `datalad wtf` could stand for many things. "Why the Face?" "Wow, that's fantastic!", "What's this for?", "What to fix", "What the FAQ", "Where's the fire?", "Wipe the floor", "Welcome to fun", "Waste Treatment Facility", "What's this foolishness", "What the fruitcake", ... Pick a translation of your choice and make running this command more joyful.

This lengthy output will report all information that might be relevant – from DataLad to GIT-ANNEX or Python up to your operating system.

The second step, finding and reading the help page of the command in question, is important in order to find out how the command that failed is used. Are arguments specified correctly? Does the help list any caveats?

There are multiple ways to find help on DataLad commands. You could turn to the [documentation](#)[314]. Alternatively, to get help right inside the terminal, run any command with the -h/--help option (also shown as an excerpt here):

```
$ datalad get --help
Usage: datalad get [-h] [-s LABEL] [-d PATH] [-r] [-R LEVELS] [-n]
                   [-D DESCRIPTION] [--reckless [auto|ephemeral|shared-...]]
                   [-J NJOBS]
                   [PATH ...]

Get any dataset content (files/directories/subdatasets).

This command only operates on dataset content. To obtain a new independent
dataset from some source use the CLONE command.

By default this command operates recursively within a dataset, but not
across potential subdatasets, i.e. if a directory is provided, all files in
the directory are obtained. Recursion into subdatasets is supported too. If
enabled, relevant subdatasets are detected and installed in order to
fulfill a request.

Known data locations for each requested file are evaluated and data are
obtained from some available location (according to git-annex configuration
and possibly assigned remote priorities), unless a specific source is
specified.

*Getting subdatasets*

Just as DataLad supports getting file content from more than one location,
the same is supported for subdatasets, including a ranking of individual
sources for prioritization.

NOTE
  Power-user info: This command uses git annex get to fulfill
  file handles.

*Examples*

Get a single file::

   % datalad get <path/to/file>

Get contents of a directory::

   % datalad get <path/to/dir/>

Get all contents of the current dataset and its subdatasets::

   % datalad get . -r
```

(continues on next page)

---

[314] http://docs.datalad.org/

```
Get (clone) a registered subdataset, but don't retrieve data::

  % datalad get -n <path/to/subds>

*Arguments*
  PATH                    path/name of the requested dataset component. The
                          component must already be known to a dataset. To add
                          new components to a dataset use the ADD command.
                          Constraints: value must be a string

*Options*
  -h, --help, --help-np
                          show this help message. --help-np forcefully disables
                          the use of a pager for displaying the help message
  -s LABEL, --source LABEL
                          label of the data source to be used to fulfill
                          requests. This can be the name of a dataset sibling or
                          another known source. Constraints: value must be a
                          string
```

This for example is the help page on **datalad get** (the same you would find in the documentation, but in your terminal, here - for brevity - slightly cut). It contains a command description, a list of all the available options with a short explanation of them, and example commands. The paragraph *Options* shows all optional flags, and all required parts of the command are listed in the paragraph *Arguments*. One first thing to check for example is whether your command call specified all of the required arguments.

Asking questions (right)

If nothing you do on your own helps to solve the problem, consider asking others. Check out neurostars[315] and search for your problem – likely, somebody already encountered the same error before[316] and fixed it, but if not, just ask a new question with a datalad tag.

Make sure your question is as informative as it can be for others. Include

- *context* – what did you want to do and why?

- the *problem* – paste the error message (all of it), and provide the steps necessary to reproduce it.

- *technical details* – what version of DataLad are you using, what version of git-annex, and which git-annex repository type, what is your operating system and – if applicable – Python version? **datalad wtf** is your friend to find all of this information.

The "submit a question link" on DataLad's GitHub page[317] page prefills a neurostars form with a

---

[315] https://neurostars.org/
[316] http://imgs.xkcd.com/comics/wisdom_of_the_ancients.png
[317] https://neurostars.org/new-topic?body=-%20Please%20describe%20the%20problem.%0A-%20What%20steps%20will%20reproduce%20the%20problem%3F%0A-%20What%20version%20of%20DataLad%20are%20you%20using%20%28run%20%60datalad%20--version%60%29%3F%20On%20what%20operating%20system%20%28consider%20running%20%60datalad%20plugin%20wtf%60%29%3F%0A-%20Please%20provide%20any%20additional%20information%20below.%0A-%20Have%20you%20had%20any%20luck%20using%20DataLad%20before%3F%20%28Sometimes%20we%20get%20tired%20of%20reading%20bug%20reports%20all%20day%20and%20a%20lil%27%20positive%20end%20note%20does%20wonders%29&tags=datalad

template for a question for a good starting point if you want to have more guidance or encounter writer's block.

Debugging like a DataLad-developer

If you have read a command's help from start to end, checked all software versions twice, even asked colleagues to reproduce your problem (unsuccessfully)[318], and you still don't have any clue what is going on, then welcome to the debugging section!

Fig. 14.1: It's not as bad as this

It is not always straightforward to see *why* a particular DataLad command has failed. Given that operations with DataLad can be quite complicated, and could involve complexities such as different forms of authentication, different file systems, interactions with the environment, configurations, and other software, and *much* more, there are what may feel like an infinite amount of sources for the problem at hand. The resulting error message, however, may not display the underlying cause correctly because the error message of whichever process failed is not propagated into the final result report. Thus, you may end up with an uninformative `Unable to access these remotes` error in the result summary, when the underlying issue is a certificate error[319].

In situations where there is no obvious reason for a command to fail, it can be helpful – either for yourself or for further information to paste into GITHUB issues – to start debugging[320], or *logging at a higher granularity* than is the default. This allows you to gain more insights into the actions DataLad and its underlying tools are taking, where *exactly* they fail, and to even play around with the program at the state of the failure.

DEBUGGING and LOGGING are not as complex as these terms may sound if you have never consciously debugged. Procedurally, it can be as easy as adding an additional flag to a command call, and cognitively, it can be as easy as engaging your visual system in a visual search task for the color red or the word "error", or reading more DataLad output that you're used to. The paragraphs below start with the general concepts, and collect concrete debugging strategies for different problems. If you have advice to add, please get in touch[321].

---

[318] https://xkcd.com/2083/

[319] https://github.com/datalad/datalad/issues/4651#issuecomment-649180205

[320] https://xkcd.com/1722/

[321] https://github.com/datalad-handbook/book/issues

Logging

In order to gain more insights into the steps performed by a program and capture as many details as possible for troubleshooting an error, you can turn to LOGGING. Logging simply refers to the fact that DataLad and its underlying tools tell you what they are doing: This information can be coarse, such as a mere `[INFO] Downloading <some_url> into <some_target>`, or fine-grained, such as `[DEBUG] Resolved dataset for status reporting: <dataset>`. The LOG LEVEL in brackets at the beginning of the line indicates how many details DataLad shares with you.

Note that LOGGING is not a sealed book, and happens automatically during the execution of any DataLad command. While you were reading the handbook, you have seen a lot of log messages already. Anything printed to your terminal preceded by `[INFO]`, for example, is a log message (in this case, on the info level). When you are *consciously* logging, you simply set the log-level to the desired amount of information, or increase the amount of verbosity until the output gives you a hint of what went wrong. Likewise, adjusting the log-level also works the other way around, and lets you *decrease* the amount of information you receive in your terminal.

> **M14.8 Log levels**
>
> Log levels provide the means to adjust how much information you want, and are described in human readable terms, ordered by the severity of the failures or problems reported. The following log levels can be chosen from:
> - **critical**: Only catastrophes are reported. Currently, there is nothing inside of DataLad that would log at this level, so setting the log level to *critical* will result in getting no details at all, not even about errors or failures.
> - **error**: With this log level you will receive reports on any errors that occurred within the program during command execution.
> - **warning**: At this log level, the command execution will report on usual situations and anything that *might* be a problem, in addition to report anything from the *error* log level. .
> - **info**: This log level will include reports by the program that indicate normal behavior and serve to keep you up to date about the current state of things, in additions to warning and error logging messages.
> - **debug**: This log level is very useful to troubleshoot a problem, and results in DataLad telling you *a lot* it possibly can.
>
> Other than log *levels*, you can also adjust the amount of information provided with numerical granularity. Instead of specifying a log level, provide an integer between 1 and 9, with lower values denoting more debugging information.
>
> Raising the log level (e.g, to `error`, or 9) will decrease the amount of information and output you will receive, while lowering it (e.g., to `debug` or 1) will increase it.

Setting a log level can be done in the form of an ENVIRONMENT VARIABLE, a configuration, or with the `-l/--log-level` flag appended directly after the main **datalad** command. To get extensive information on what **datalad status** does underneath the hood, your command could look like this:

```
$ datalad --log-level debug status
[DEBUG] Command line args 1st pass for DataLad 0.14.0rc1.dev10-g1b221. Parsed:␣
→Namespace() Unparsed: ['status']
[DEBUG] Discovering plugins
[DEBUG] Building doc for <class 'datalad.core.local.status.Status'>
```

(continues on next page)

```
[DEBUG] Parsing known args among ['/home/adina/env/handbook2/bin/datalad', '--log-level',
↪'debug', 'status']
[DEBUG] Async run ['git', '--git-dir=', 'config', '-z', '-l', '--show-origin']
[DEBUG] Launching process ['git', '--git-dir=', 'config', '-z', '-l', '--show-origin']
[DEBUG] Process 1207466 started
[DEBUG] Waiting for process 1207466 to complete
[DEBUG] Process 1207466 exited with return code 0
[DEBUG] Determined class of decorated function: <class 'datalad.core.local.status.Status'>
↪
[DEBUG] Resolved dataset for status reporting: /home/me/dl-101/DataLad-101
[DEBUG] Async run ['git', 'config', '-z', '-l', '--show-origin']
[DEBUG] Launching process ['git', 'config', '-z', '-l', '--show-origin']
[DEBUG] Process 1207468 started
[DEBUG] Async run ['git', 'ls-files', '-z', '-m']
[DEBUG] Launching process ['git', 'ls-files', '-z', '-m']
[DEBUG] Process 1207491 started
[DEBUG] Waiting for process 1207491 to complete
[DEBUG] Process 1207491 exited with return code 0
[DEBUG] AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project).get_content_info(...)
[DEBUG] Query repo: ['ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Async run ['git', 'ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Launching process ['git', 'ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Process 1207493 started
[DEBUG] Waiting for process 1207493 to complete
[DEBUG] Process 1207493 exited with return code 0
[DEBUG] Done query repo: ['ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Done AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project).get_content_info(...)
[DEBUG] Async run ['git', 'config', '-z', '-l', '--show-origin']
[DEBUG] Launching process ['git', 'config', '-z', '-l', '--show-origin']
[DEBUG] Process 1207495 started
[DEBUG] Waiting for process 1207495 to complete
[DEBUG] Process 1207495 exited with return code 0
[DEBUG] Async run ['git', 'config', '-z', '-l', '--show-origin', '--file', '/home/me/dl-
↪101/DataLad-101/midterm_project/input/.datalad/config']
[DEBUG] Launching process ['git', 'config', '-z', '-l', '--show-origin', '--file', '/home/
↪me/dl-101/DataLad-101/midterm_project/input/.datalad/config']
[DEBUG] Process 1207497 started
[DEBUG] Waiting for process 1207497 to complete
[DEBUG] Process 1207497 exited with return code 0
[DEBUG] Async run ['git', 'symbolic-ref', 'HEAD']
[DEBUG] Launching process ['git', 'symbolic-ref', 'HEAD']
[DEBUG] Process 1207499 started
[DEBUG] Waiting for process 1207499 to complete
[DEBUG] Process 1207499 exited with return code 0
[DEBUG] Async run ['git', 'rev-parse', '--quiet', '--verify', 'HEAD^{commit}']
[DEBUG] Launching process ['git', 'rev-parse', '--quiet', '--verify', 'HEAD^{commit}']
[DEBUG] Process 1207501 started
[DEBUG] Waiting for process 1207501 to complete
[DEBUG] Process 1207501 exited with return code 0
[DEBUG] AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project/input).get_content_info(...)
[DEBUG] Query repo: ['ls-files', '--stage', '-z', '--exclude-standard', '-o', '--directory
↪', '--no-empty-directory']
[DEBUG] Async run ['git', 'ls-files', '--stage', '-z', '--exclude-standard', '-o', '--
↪directory', '--no-empty-directory']
[DEBUG] Launching process ['git', 'ls-files', '--stage', '-z', '--exclude-standard', '-o',
↪ '--directory', '--no-empty-directory']
```

```
[DEBUG] Process 1207503 started
[DEBUG] Waiting for process 1207503 to complete
[DEBUG] Process 1207503 exited with return code 0
[DEBUG] Done query repo: ['ls-files', '--stage', '-z', '--exclude-standard', '-o', '--
→directory', '--no-empty-directory']
[DEBUG] Done AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project/input).get_content_
→info(...)
[DEBUG] Async run ['git', 'ls-files', '-z', '-m']
[DEBUG] Launching process ['git', 'ls-files', '-z', '-m']
[DEBUG] Process 1207505 started
[DEBUG] Waiting for process 1207505 to complete
[DEBUG] Process 1207505 exited with return code 0
[DEBUG] AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project/input).get_content_info(...)
[DEBUG] Query repo: ['ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Async run ['git', 'ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Launching process ['git', 'ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Process 1207507 started
[DEBUG] Waiting for process 1207507 to complete
[DEBUG] Process 1207507 exited with return code 0
[DEBUG] Done query repo: ['ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Done AnnexRepo(/home/me/dl-101/DataLad-101/midterm_project/input).get_content_
→info(...)
[DEBUG] Async run ['git', 'config', '-z', '-l', '--show-origin']
[DEBUG] Launching process ['git', 'config', '-z', '-l', '--show-origin']
[DEBUG] Process 1207509 started
[DEBUG] Waiting for process 1207509 to complete
[DEBUG] Process 1207509 exited with return code 0
[DEBUG] Async run ['git', 'config', '-z', '-l', '--show-origin', '--file', '/home/me/dl-
→101/DataLad-101/recordings/longnow/.datalad/config']
[DEBUG] Launching process ['git', 'config', '-z', '-l', '--show-origin', '--file', '/home/
→me/dl-101/DataLad-101/recordings/longnow/.datalad/config']
[DEBUG] Process 1207511 started
[DEBUG] Waiting for process 1207511 to complete
[DEBUG] Process 1207511 exited with return code 0
[DEBUG] Async run ['git', 'symbolic-ref', 'HEAD']
[DEBUG] Launching process ['git', 'symbolic-ref', 'HEAD']
[DEBUG] Process 1207513 started
[DEBUG] Waiting for process 1207513 to complete
[DEBUG] Process 1207513 exited with return code 0
[DEBUG] Async run ['git', 'rev-parse', '--quiet', '--verify', 'HEAD^{commit}']
[DEBUG] Launching process ['git', 'rev-parse', '--quiet', '--verify', 'HEAD^{commit}']
[DEBUG] Process 1207515 started
[DEBUG] Waiting for process 1207515 to complete
[DEBUG] Process 1207515 exited with return code 0
[DEBUG] AnnexRepo(/home/me/dl-101/DataLad-101/recordings/longnow).get_content_info(...)
[DEBUG] Query repo: ['ls-files', '--stage', '-z', '--exclude-standard', '-o', '--directory
→', '--no-empty-directory']
[DEBUG] Async run ['git', 'ls-files', '--stage', '-z', '--exclude-standard', '-o', '--
→directory', '--no-empty-directory']
[DEBUG] Launching process ['git', 'ls-files', '--stage', '-z', '--exclude-standard', '-o',
→ '--directory', '--no-empty-directory']
[DEBUG] Process 1207517 started
[DEBUG] Waiting for process 1207517 to complete
[DEBUG] Process 1207517 exited with return code 0
[DEBUG] Done query repo: ['ls-files', '--stage', '-z', '--exclude-standard', '-o', '--
→directory', '--no-empty-directory']
```

```
[DEBUG] Done AnnexRepo(/home/me/dl-101/DataLad-101/recordings/longnow).get_content_info(..
↪.)
[DEBUG] Async run ['git', 'ls-files', '-z', '-m']
[DEBUG] Launching process ['git', 'ls-files', '-z', '-m']
[DEBUG] Process 1207519 started
[DEBUG] Waiting for process 1207519 to complete
[DEBUG] Process 1207519 exited with return code 0
[DEBUG] AnnexRepo(/home/me/dl-101/DataLad-101/recordings/longnow).get_content_info(...)
[DEBUG] Query repo: ['ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Async run ['git', 'ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Launching process ['git', 'ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Process 1207521 started
[DEBUG] Waiting for process 1207521 to complete
[DEBUG] Process 1207521 exited with return code 0
[DEBUG] Done query repo: ['ls-tree', 'HEAD', '-z', '-r', '--full-tree', '-l']
[DEBUG] Done AnnexRepo(/home/me/dl-101/DataLad-101/recordings/longnow).get_content_info(..
↪.)
nothing to save, working tree clean
```

> 💡 **M14.9 ... and how does it look when using environment variables or configurations?**
>
> The log level can also be set (for different scopes) using the datalad.log.level configuration variable, or the corresponding environment variable DATALAD_LOG_LEVEL.
> To set the log level for a single command, for example, set it in front of the command:
>
> ```
> $ DATALAD_LOG_LEVEL=debug datalad status
> ```
>
> And to set the log level for the rest of the shell session, export it:
>
> ```
> $ export DATALAD_LOG_LEVEL=debug
> $ datalad status
> $ ...
> ```
>
> You can find out a bit more on environment variable *in this Findoutmore* (page 125).
> The configuration variable can be used to set the log level on a user (global) or system-wide level with the **git config** command:
>
> ```
> $ git config --global datalad.log.level debug
> ```

This output is extensive and detailed, but it precisely shows the sequence of commands and arguments that are run prior to a failure or crash, and all additional information that is reported with the log levels info or debug can be very helpful to find out what is wrong. Even if the vast amount of detail in output generated with debug logging appears overwhelming, it can make sense to find out at which point an execution stalls, whether arguments, commands, or datasets reported in the debug output are what you expect them to be, and to forward all information into any potential GitHub issue you will be creating.

Finally, other than logging with a DataLad command, it sometimes can be useful to turn to GIT-ANNEX or GIT for logging. For failing **datalad get** calls, it may be useful to retry the retrieval using **git annex get -d -v <file>**, where –d (debug) and –v (verbose) increase the amount of detail about the command execution and failure. In rare cases where you suspect something might be wrong with Git, setting the environment variables GIT_TRACE and GIT_TRACE_SETUP to 2 prior to running a Git command will give you debugging output.

Debugging

If the additional level of detail provided by logging messages is not enough, you can go further with actual DEBUGGING. For this, add the `--dbg` or `--idbg` flag to the main **datalad** command, as in `datalad --dbg status`. Adding this flag will enter a Python[322] or IPython debugger[323] when something unexpectedly crashes. This allows you to debug the program right when it fails, inspect available variables and their values, or step back and forth through the source code. Note that debugging experience is not a prerequisite when using DataLad – although it is an exciting life skill[324]. The official Python docs[325] provide a good overview on the available debugger commands if you are interested in learning more about this.

Debugging examples

This section collects errors and their solutions from real GitHub issues. They may not be applicable for the problem you are currently facing, but seeing other's troubleshooting strategies may be helpful nevertheless. If you are interested in getting your error and solution described here, please get in touch[326].

datalad get: It is common for **datalad get** errors to originate in GIT-ANNEX, the software used by DataLad to transfer data. Here are a few suggestions to debug them:

- Take a deep breath, or preferably a walk in a nice park :)

- **Check that you are using a recent version of git-annex**

  - `git-annex version` returns the version of git-annex on the first line of its input, and it is also reported in the output of **datalad wtf**.

  - The version number contains the release date of the version in use. For instance, git-annex version: `8.20200330-g971791563` was released on 30 March 2020.

  - If the version that you are using is older than a few months, consider updating using the instructions here[327].

- Try to download the file using `git-annex get -v -d <file_name>`. If this doesn't succeed, the DataLad command may not succeed. Options `-d/--debug` and `-v` are here to provide as much verbosity in error messages as possible

- Read the output of GIT-ANNEX, identify the error, breathe again, and solve the issue! Table 14.1 list a few common or tricky ones.

Table 14.1: Examples of possible git-annex issues.

---

[322] https://docs.python.org/3/library/pdb.html
[323] https://ipython.org/
[324] https://www.monkeyuser.com/2017/step-by-step-debugging/
[325] https://docs.python.org/3/library/pdb.html#debugger-commands
[326] https://github.com/datalad-handbook/book/issues
[327] http://handbook.datalad.org/en/latest/intro/installation.html

git-annex error	A solution that worked once
	Unset environment variable `CURL_CA_BUNDLE`
```Last exception was:Could not find a suitable TLS CAcertificate bundle, invalid path:/etc/pki/tls/certs/ca-bundle.crt[adapters.py:cert_verify:227]```	
Permission denied when writing file	Download to a writeable file system
File retrieval from an Amazon S3 bucket failed during a system call in a MATLAB session: ```>> system('datalad -C mytest \ get 100206/T1w/T1w_acpc_dc.nii.↪gz') [...] git-annex: get: 1 failed```	MATLAB massively overrides the `LD_LIBRARY_PATH` setting. This can lead to a number of issues, among them SSL certification errors. Prefixing the **datalad get** command with `!LD_LIBRARY_PATH= datalad get [....]` can solve this.

Common warnings and errors

A lot of output you will see while working with DataLad originates from warnings or errors by DataLad, git-annex, or Git. Some of these outputs can be wordy and not trivial to comprehend - and even if everything works, some warnings can be hard to understand. This following section will list some common git-annex warnings and errors and attempts to explain them. If you encounter warnings or errors that you would like to see explained in this book, please let us know by filing an issue[328].

Output produced by Git

Unset Git identity

If you have not configured your Git identity, you will see warnings like this when running any DataLad command:

```
[WARNING] It is highly recommended to configure git first (set both user.name and user.↪email) before using DataLad.
```

To set your Git identity, go back to section *Initial configuration* (page 20).

Rejected pushes

One error you can run into when publishing dataset contents is that your **datalad push** to a sibling is rejected. One example is this:

```
$ datalad push --to public
 [ERROR  ] refs/heads/master->public:refs/heads/master [rejected] (non-fast-forward)↪[publish(/home/me/dl-101/DataLad-101)]
```

This example is an attempt to push a local dataset to its sibling on GitHub. The push is rejected because it is a non-fast-forward merge situation. Usually, this means that the sibling contains

---

[328] https://github.com/datalad-handbook/book/issues/new

changes that your local dataset does not yet know about. It can be fixed by updating from the sibling first with a `datalad update --merge`.

Here is a different push rejection:

```
$ datalad push --to roommate
 publish(ok): . (dataset) [refs/heads/git-annex->roommate:refs/heads/git-annex 023a541..
↪59a6f8d]
 [ERROR  ] refs/heads/master->roommate:refs/heads/master [remote rejected] (branch is␣
↪currently checked out) [publish(/home/me/dl-101/DataLad-101)]
 publish(error): . (dataset) [refs/heads/master->roommate:refs/heads/master [remote␣
↪rejected] (branch is currently checked out)]
 action summary:
   publish (error: 1, ok: 1)
```

As you can see, the GIT-ANNEX BRANCH was pushed successfully, but updating the `master` branch was rejected: `[remote rejected] (branch is currently checked out) [publish(/home/me/dl-101/DataLad-101)]`. In this particular case, this is because it was an attempt to push from `DataLad-101` to the `roommate` sibling that was created in chapter *Collaboration* (page 92). This is a special case of pushing, because it – in technical terms – is a push to a non-bare repository. Unlike BARE GIT REPOSITORIES, non-bare repositories can not be pushed to at all times. To fix this, you either want to checkout another branch[329] in the `roommate` sibling or push to a non-checked out branch in the `roommate` sibling. Alternatively, you can configure `roommate` to receive the push with Git's `receive.denyCurrentBranch` configuration key. By default, this configuration is set to `refuse`. Setting it to `updateInstead` with `git config receive.denyCurrentBranch updateInstead` will allow updating the checked out branch. See `git config`s man page entry[330] on `receive.denyCurrentBranch` for more.

Detached HEADs

One warning that you may encounter during an installation of a dataset is:

```
[INFO   ] Submodule HEAD got detached. Resetting branch master to point to 046713bb.␣
↪Original location was 47e53498
```

Even though "detached HEAD" sounds slightly worrisome, this is merely an information and does not require an action from your side. It is related to Git submodules[331] (the underlying Git concept for subdatasets), and informs you about the current state a subdataset is saved in the superdataset you have just cloned.

Output produced by git-annex

Unusable annexes

Upon installation of a dataset, you may see:

```
[INFO   ]     Remote origin not usable by git-annex; setting annex-ignore
[INFO   ]     This could be a problem with the git-annex installation on the
remote. Please make sure that git-annex-shell is available in PATH when you
ssh into the remote. Once you have fixed the git-annex installation,
run: git annex enableremote origin
```

---

[329] https://git-scm.com/docs/git-checkout
[330] https://git-scm.com/docs/git-config#Documentation/git-config.txt-receivedenyCurrentBranch
[331] https://git-scm.com/book/en/v2/Git-Tools-Submodules

This warning lets you know that git-annex will not attempt to download content from the REMOTE "origin". This can have many reasons, but as long as there are other remotes you can access the data from, you are fine.

A similar warning message may appear when adding a sibling that is a pure Git REMOTE, for example a repository on GitHub:

```
[INFO ] Failed to enable annex remote github, could be a pure git or not
accessible
[WARNING] Failed to determine if github carries annex. Remote was marked by
annex as annex-ignore. Edit .git/config to reset if you think that was done
by mistake due to absent connection etc
```

These messages indicate that the sibling github does not carry an annex. Thus, annexed file contents can not be pushed to this sibling. This happens if the sibling indeed does not have an annex (which would be true, for example, for siblings on GitHub, GitLab, Bitbucket, ..., and would not require any further action or worry), or if the remote could not be reached, e.g., due to a missing internet connection (in which case you could set the key annex-ignore in .git/config to false).

Speaking of remotes that are not available, this will probably be one of the most commonly occurring git-annex errors to see - failing **datalad get** calls because remotes are not available:

Other errors

Sometimes, registered subdatasets URLs have an SSH instead of HTTPS address, for example git@github.com:datalad-datasets/longnow-podcasts.git instead of https://github.com/datalad-datasets/longnow-podcasts.git. If one does not have an SSH key configured for the required service (e.g., GitHub, or a server), installing or getting the subdataset and its contents fails, with messages starting similar to this:

```
[INFO   ] Cloning https://github.com/psychoinformatics-de/paper-remodnav.git/remodnav [2
↪other candidates] into '/home/homeGlobal/adina/paper-remodnav/remodnav'
Permission denied (publickey).
```

If you encounter these errors, make sure to create and/or upload an SSH key (see section *Walkthrough: Dataset hosting on GIN* (page 208) for an example) as necessary, or reconfigure/edit the URL into a HTTPS URL.

git-annex as the default branch on GitHub

If you publish a dataset to GitHub, but the resulting repository seems to consist of cryptic directories instead of your actual file names and directories, GitHub may have made the GIT-ANNEX BRANCH the default.

To find out more about this and how to fix it, please take a look at the corresponding *FAQ* (page 487).

Windows adds whitespace line-endings to unchanged files

The type of line ending (a typically invisible character that indicates a line break) differs between operating system. While Linux and OSX use a *line feed* (LF), Windows uses *carriage return + line feed* (CRLF). When you only collaborate across operating systems of the same type, this is a very boring fun fact at most. But if Windows- and Non-Windows users collaborate, or if you are working with files across different operating systems, the different type of line ending that Windows uses may show up as unintended modifications on other system. In most cases, this is

prevented by a default cross-platform compatible line-ending configuration on Windows that is set during installation:
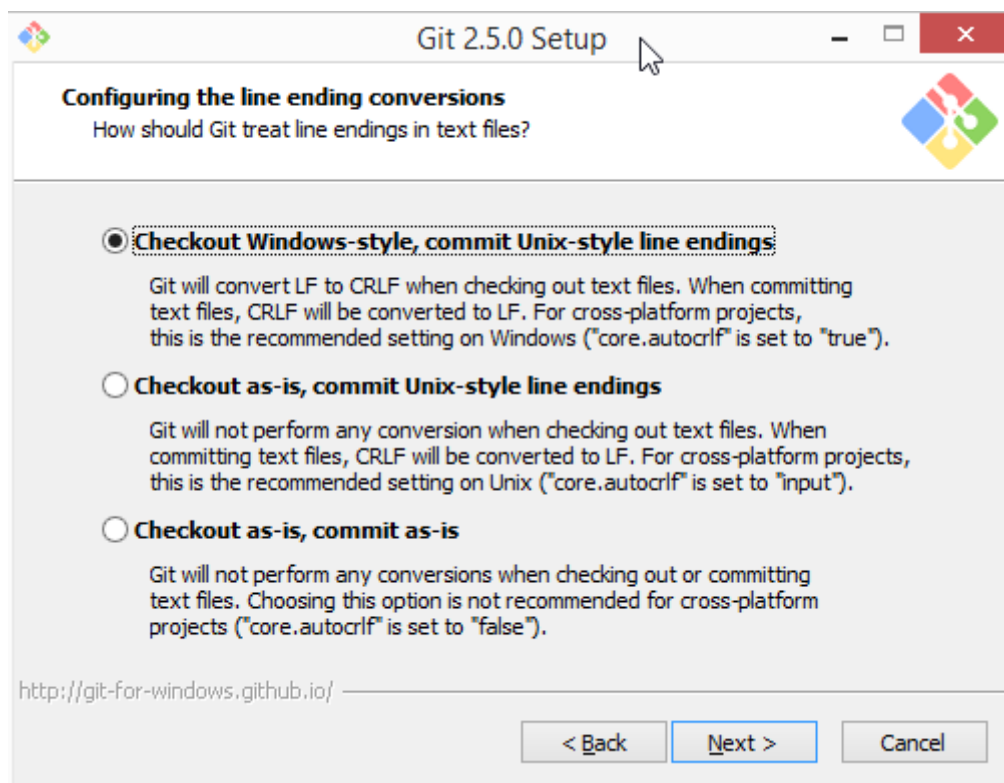
To fix this behavior outside of the installation process and standardize line endings across operating systems, Windows users are advised to set the configuration `core.autcrlf true` with `git config --global core.autocrfl true`.

asyncio errors at DataLad import

DataLad's internal command runner uses asyncio[332]. This can lead to an error when DataLad is used within a script or application that itself uses asyncio. To summarize the problem with a quote from Python's bug tracker[333]: "you can't call async code from sync code that's being called from async code".

Jupyter Notebooks are probably the most likely place that you'll run into this error (ipykernel issue 548[334]). When importing `datalad`, you'll see this:

```
RuntimeError: Cannot run the event loop while another loop is running
```

The nest-asyncio[335] package provides a workaround:

```
>>> import nest_asyncio
>>> nest_asyncio.apply()

>>> import datalad
```

---

[332] https://docs.python.org/3/library/asyncio.html
[333] https://bugs.python.org/issue33523#msg349561
[334] https://github.com/ipython/ipykernel/issues/548
[335] https://pypi.org/project/nest-asyncio/

14.5 Gists

The more complex and larger your DataLad project, the more difficult it is to do efficient house-keeping. This section is a selection of code snippets tuned to perform specific, non-trivial tasks in datasets. Often, they are not limited to single commands of the version control tools you know, but combine helpful other command line tools and general Unix command line magic. Just like GitHub gists[337], its a collection of lightweight and easily accessible tips and tricks. For a more basic command overview, take a look at the *DataLad cheat sheet* (page 496). The tips collection of git-annex[338] is also a very valuable resource.

If there are tips you want to share, or if there is a question you would like to see answered here, please get in touch[339].

Parallelize subdataset processing

DataLad can not yet parallelize processes that are performed independently over a large number of subdatasets. Pushing across a dataset hierarchy or creating RIA siblings for all subdatasets of a superdataset, for example, is performed one after the other. Unix however, has a few tools such as xargs[340] or the `parallel` tool of moreutils[341] that can assist.

Here is an example of pushing all subdatasets (and their respective subdatasets) recursively to their (identically named) siblings:

```
$ datalad -f '{path}' subdatasets | xargs -n 1 -P 10 datalad push -r --to <sibling-name> -
↪d
```

`datalad -f '{path}' subdatasets` discovers the paths of all subdatasets, and `xargs` hands them individually (`-n 1`) to a (recursive) **datalad push**, but performs 10 of these operations in parallel (`-P 10`), thus achieving parallelization.

Here is an example of cross-dataset download parallelization:

```
$ datalad -f '{path}' subdatasets | xargs -n 1 -P 10 datalad get -d
```

---

[337] https://gist.github.com/
[338] https://git-annex.branchable.com/tips/
[339] https://github.com/datalad-handbook/book/issues/new
[340] https://en.wikipedia.org/wiki/Xargs
[341] https://joeyh.name/code/moreutils/

Operations like this can safely be attempted for all commands that are independent across subdatasets.

Check whether all file content is present locally

In order to check if all the files in a dataset have their file contents locally available, you can ask git-annex:

```
$ git annex find --not --in=here
```

Any file that does not have its contents locally available will be listed. If there are subdatasets you want to recurse into, use the following command

```
$ git submodule foreach --quiet --recursive \
 'git annex find --not --in=here --format=$displaypath/$\\{file\\}\\n'
```

Alternatively, to get very comprehensive output, you can use

```
$ datalad -f json status --recursive --annex availability
```

The output will be returned as json, and the key `has_content` indicates local content availability (`true` or `false`). To filter through it, the command line tool jq[342] works well:

```
$ datalad -f json status --recursive --annex all | jq '. | select(.has_content == true).
↪path'
```

Drop annexed files from all past commits

If there is annexed file content that is not used anymore (i.e., data in the annex that no files in any branch point to anymore such as corrupt files), you can find out about it and remove this file content out of your dataset (i.e., completely and irrecoverably delete it) with git-annex's commands **`git annex unused`** and **`git annex dropunused`**.

Find out which file contents are unused (not referenced by any current branch):

```
$ git annex unused
 unused . (checking for unused data...)
   Some annexed data is no longer used by any files in the repository.
     NUMBER  KEY
       1        SHA256-s86050597--
↪6ae2688bc533437766a48aa19f2c06be14d1bab9c70b468af445d4f07b65f41e
       2        SHA1-s14--f1358ec1873d57350e3dc62054dc232bc93c2bd1
   (To see where data was previously used, try: git log --stat -S'KEY')
   (To remove unwanted data: git-annex dropunused NUMBER)
 ok
```

Remove a single unused file by specifying its number in the listing above:

```
$ git annex dropunused 1
 dropunused 1 ok
```

Or a range of unused data with

---

[342] https://stedolan.github.io/jq/

```
$ git annex dropunused 1-1000
```

Or all

```
$ git annex dropunused all
```

Getting single file sizes prior to downloading from the Python API and the CLI

For a single file, **datalad status --annex -- myfile** will report on the size of the file prior to a **datalad get**.

If you want to do it in Python, try this approach:

```python
import datalad.api as dl

ds = dl.Dataset("/path/to/some/dataset")
results = ds.status(path=<path or list of paths>, annex="basic", result_renderer=None)
```

Check whether a dataset contains an annex

Datasets can be either GitRepos (i.e., sole Git repositories; this happens when they are created with the --no-annex flag, for example), or AnnexRepos (i.e., datasets that contain an annex). Information on what kind of repository it is is stored in the dataset report of **datalad wtf** under the key repo. Here is a one-liner to get this info:

```
$ datalad -f'{infos[dataset][repo]}' wtf
```

Backing-up datasets

In order to back-up datasets you can publish them to a REMOTE INDEXED ARCHIVE (RIA) STORE or to a sibling dataset. The former solution does not require Git, git-annex, or DataLad to be installed on the machine that the back-up is pushed to, the latter does require them.

To find out more about RIA stores, checkout the section *Remote Indexed Archives for dataset storage and backup* (page 294). A sketch of how to implement a sibling for backups is below:

```
# create a back up sibling
datalad create-sibling --annex-wanted anything -r myserver:/path/to/backup
# publish a full backup of the current branch
datalad publish --to=myserver -r
# subsequently, publish updates to be backed up with
datalad publish --to=myserver -r --since= --missing=inherit
```

In order to push not only the current branch, but refs, add the option --publish-by-default "refs/*" to the **create-sibling** call. Should you want to back up all annexed data, even past versions of files, use **git annex sync** to push to the sibling:

```
$ git annex sync --all --content <sibling-name>
```

For an in-depth explanation and example take a look at the GitHub issue that raised this question[343].

Retrieve partial content from a hierarchy of (uninstalled) datasets

In order to **get** dataset content across a range of subdatasets, a bit of UNIX command line foo can increase the efficiency of your command.

Example: consider retrieving all `ribbon.nii.gz` files for all subjects in the HCP open access dataset[344] (a dataset with about 4500 subdatasets – read on more about it in *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433)). If all subject-subdatasets are installed (e.g., with `datalad get -n -r` for a recursive installation without file retrieval), GLOBBING with the shell works fine:

```
$ datalad get HCP1200/*/T1W/ribbon.nii.gz
```

The Gist *Parallelize subdataset processing* (page 273) can show you how to parallelize this. If the subdatasets are not yet installed, globbing will not work, because the shell can't expand non-existent paths. As an alternative, you can pipe the output of an (arbitrarily complex) **datalad search** command into **datalad get**:

```
$ datalad -f '{path}' -c datalad.search.index-egrep-documenttype=all search 'path:.*T1w.
↪*\.nii.gz' | xargs -n 100 datalad get
```

However, if you know the file locations within the dataset hierarchy and they are predictably named and consistent, you can create a file containing all paths to be retrieved and pipe that into **get** as well:

```
# create file with all file paths
$ for sub in HCP1200/*; do echo ${sub}/T1w/ribbons.nii.gz; done > toget.txt
# pipe it into datalad get
$ cat toget.txt | xargs -n 100 datalad get
```

Speed up status reports in large datasets

In datasets with deep dataset hierarchies or large numbers of files, **datalad status** calls can be expensive. Handily, the command provides options that can boost performance by limiting what is being tested and reported. In order to speed up subdataset state state evaluation, `-e/--eval-subdataset-state` can be set `commit` or `no`. Instead of checking recursively for uncommitted modifications in subdatasets, this would lead `status` to only compare the most recent commit SHASUM in the subdataset against the recorded subdataset state in the superdataset (`commit`), or skip subdataset state evaluation completely (`no`). In order to speed up file type evaluation, the option `-t/--report-filetype` can be set to `raw`. This skips an evaluation on whether symlinks are pointers to annexed file (upon which, if true, the symlink would be reported as type "file"). Instead, all symlinks will be reported as being of type "symlink".

---

[343] https://github.com/datalad/datalad/issues/4369
[344] https://github.com/datalad-datasets/human-connectome-project-openaccess

Squashing git-annex history

A large number of commits in the GIT-ANNEX BRANCH (think: thousands rather than hundreds) can inflate your repository and increase the size of the .git directory, which can lead to slower cloning operations. There are, however, ways to shrink the commit history in the annex branch.

In order to SQUASH the entire git-annex history into a single commit, run

```
$ git annex forget --drop-dead --force
```

Afterwards, if your dataset has a sibling, the branch needs to be FORCE-PUSHed. If you attempt an operation to shrink your git-annex history, also checkout this thread[345] for more information on shrinking git-annex's history and helpful safeguards and potential caveats.

---

[345] https://git-annex.branchable.com/forum/safely_dropping_git-annex_history/

M14.5 Changing the commit messages of not-the-most-recent commits

The `git commit --amend` command will let you rewrite the commit message of the most recent commit. If you however need to rewrite commit messages of older commits, you can do so during a so-called "interactive rebase"[307]. The command for this is

```
$ git rebase -i HEAD~N
```

where N specifies how far back you want to rewrite commits. `git rebase -i HEAD~3` for example lets you apply changes to the any number of commit messages within the last three commits.

Be aware that an interactive rebase lets you *rewrite* history. This can lead to confusion or worse if the history you are rewriting is shared with others, e.g., in a collaborative project. Be also aware that rewriting history that is *pushed/published* (e.g., to GitHub) will require a force-push!

Running this command gives you a list of the N most recent commits in your text editor (which may be VIM!), sorted with the most recent commit on the bottom. This is how it may look like:

```
pick 8503f26 Add note on adding siblings
pick 23f0a52 add note on configurations and git config
pick c42cba4 add note on DataLad's procedures

# Rebase b259ce8..c42cba4 onto b259ce8 (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
```

An interactive rebase allows to apply various modifying actions to any number of commits in the list. Below the list are descriptions of these different actions. Among them is "reword", which lets you "edit the commit message". To apply this action and reword the top-most commit message in this list (`8503f26 Add note on adding siblings`, three commits back in the history), exchange the word `pick` in the beginning of the line with the word `reword` or simply `r` like this:

```
r 8503f26 Add note on adding siblings
```

If you want to reword more than one commit message, exchange several `picks`. Any commit with the word `pick` at the beginning of the line will be kept as is. Once you are done, save and close the editor. This will sequentially open up a new editor for each commit you want to reword. In it, you will be able to change the commit message. Save to proceed to the next commit message until the rebase is complete. But be careful not to delete any lines in the above editor view – **An interactive rebase can be dangerous, and if you remove a line, this commit will be lost!**[308]

---

[307] Note though that rewriting history can be dangerous, and you should be aware of what you are doing. For example, rewriting parts of the dataset's history that have been published (e.g., to a GitHub repository)

already or that other people have copies of, is not advised.

[308] When in need to interactively rebase, please consult further documentation and tutorials. It is out of the scope of this handbook to be a complete guide on rebasing, and not all interactive rebasing operations are complication-free. However, you can always undo mistakes that occur during rebasing with the help of the reflog[309].

Part III

Advanced

There is more to DataLad than the Basics. Equipped with the fundamental building blocks and broad DataLad expertise, you can continue to this advanced section to read more on how DataLad can be used, from special case applications to data management at scale.

This part of the book will abandon the DataLad-101 narrative. Consider yourself graduated. There is no need to read the chapters of this book sequentially. Rather, find chapters that match your interest and usecase, and read its sections and associated usecases.

ADVANCED OPTIONS

15.1 How to hide content from DataLad

You have progressed quite far in the DataLad-101 course, and by now, you have gotten a good overview on the basics and *not-so-basic-anymore*s of DataLad. You know how to add, modify, and save files, even completely reproducibly, and how to share your work with others.

By now, the **datalad save** command is probably the most often used command in this dataset. This means that you have seen some of its peculiarities. The most striking was that it by default will save the complete datasets status if one does not provide a path to a file change. This would result in all content that is either modified or untracked being saved in a single commit.

In principle, a general recommendation may be to keep your DataLad dataset clean. This assists a structured way of working and prevents clutter, and it also nicely records provenance inside your dataset. If you have content in your dataset that has been untracked for 9 months it will be hard to remember where this content came from, whether it is relevant, and if it is relevant, for what. Adding content to your dataset will thus usually not do harm – certainly not for your dataset. However, there may be valid reasons to keep content out of DataLad's version control and tracking. Maybe you hide your secret my-little-pony-themesongs/ collection within Deathmetal/ and do not want a record of this in your history or the directory being shared together with the rest of the dataset. Who knows? We would not judge in any way.

In principle, you already know a few tricks on how to be "messy" and have untracked files. For **datalad save**, you know that precise file paths allow you to save only those modifications you want to change. For **datalad run** you know that one can specify the --explicit option to only save those modifications that are specified in the --output argument.

Beyond these tricks, there are two ways to leave *untracked* content unaffected by a **datalad save**. One is the -u/--updated option of **datalad save**:

```
$ datalad save -m "my commit message here" -u/--updated
```

will only save dataset modifications to previously tracked paths. If my-little-pony-themesongs/ is not yet tracked, a datalad save -u will leave it untouched, and its existence or content is not written to the history of your dataset.

A second way of hiding content from DataLad is a .gitignore file. As the name suggests, it is a GIT related solution, but it works just as well for DataLad.

A .gitignore file is a file that specifies which files should be *ignored* by the version control tool. To use a .gitignore file, simply create a file with this name in the root of your dataset (be mindful: remember the leading .!). You can use one of thousands of publicly shared examples[346], or create your own one.

To specify dataset content to be git-ignored, you can either write a full file name, e.g. playlists/my-little-pony-themesongs/Friendship-is-magic.mp3 into this file, or paths or patterns that make use of globbing, such as playlists/my-little-pony-themesongs/*. The hidden section at the end of this page contains some general rules for patterns in .gitignore files. Afterwards, you just need to save the file once to your dataset so that it is version controlled. If you have new content you do not want to track, you can add new paths or patterns to the file, and save these modifications.

Let's try this with a very basic example: Let's git-ignore all content in a tmp/ directory in the DataLad-101 dataset:

```
$ cat << EOT > .gitignore

tmp/*
EOT


$ datalad status
untracked: .gitignore (file)


$ datalad save -m "add something to ignore" .gitignore
add(ok): .gitignore (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

This .gitignore file is very minimalistic, but its sufficient to show how it works. If you now create a tmp/ directory, all of its contents would be ignored by your datasets version control. Let's do so, and add a file into it that we do not (yet?) want to save to the dataset's history.

```
$ mkdir tmp
$ echo "this is just noise" > tmp/a_random_ignored_file


$ datalad status
nothing to save, working tree clean
```

As expected, the file does not show up as untracked – it is being ignored! Therefore, a .gitignore file can give you a space inside of your dataset to be messy, if you want to be.

---

[346] https://github.com/github/gitignore

> 💡 **M15.1 Rules for .gitignore files**
>
> Here are some general rules for the patterns you can put into a `.gitignore` file, taken from the book Pro Git[347] :
> - Blank lines or lines starting with # are ignored
> - Standard GLOBBING patterns work. The line
>
> ```
> *.[oa]
> ```
>
> lets all files ending in `.o` or `.a` be ignored. Importantly, these patterns will be applied recursively through your dataset, so that a file matching this rule will be ignored, even if it is in a subdirectory of your dataset. If you want to ignore specific files in the directory your `.gitignore` file lies in, but not any subdirectories, start the pattern with a forward slash (/), as in `/TODO`.
> - To specify directories, you can end patterns with a forward slash (/), for example `build/`.
> - You can negate a pattern by starting it with an exclamation point (!), such as `!lib.a`. This would track the file `lib.a`, even if you would be ignoring all other files with `.a` extension.
>
> The manpage of `gitignore` has an extensive and well explained overview. To read it, simply type `man gitignore` into your terminal.
>
> You can have a single `.gitignore` file in the root of your dataset, and its rules apply recursively to the entire hierarchy of the dataset (but not subdatasets!). Alternatively, you can have additional `.gitignore` files in subdirectories of your dataset. The rules in these nested `.gitignore` files only apply to the files under the directory where they are located.
>
> _____
>
> [347] https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository#_ignoring

> ⚠️ **Implications of git-ignored outputs for re-running**
>
> Note one caveat: If a command creates an output that is git-ignored, (e.g. anything inside of `tmp/` in our dataset), a subsequent command that requires it as an undisclosed input will only succeed if both commands a ran in succession. The second command will fail if re-ran on its own, however.

> 💡 **M15.2 Globally ignoring files**
>
> Its not only possible to define files or patterns for files to ignore inside of individual datasets, but to also set global specifications to have every single dataset you own ignore certain files or file types.
>
> This can be useful, for example, for unwanted files that your operating system or certain software creates, such as lock files[348], .swp files[349], .DS_Store files[350], Thumbs.DB[351], or others.
>
> To set rules to ignore files for all of your datasets, you need to create a *global* `.gitignore` file. The only difference between a repository-specific and a global `.gitignore` file is its location on your file system. You can put it either in its default location `~/.config/git/ignore` (you may need to create the `~/.config/git` directory first), or place it into any other location and point Git to it. If you create a file at `~/.gitignore_global` and run
>
> ```
> $ git config --global core.excludesfile ~/.gitignore_global
> ```
>
> Git – and consequently DataLad – will not bother you about any of the files or file types

💡 you have specified.

---

348 https://fileinfo.com/extension/lock
349 https://www.networkworld.com/article/2931534/what-are-unix-swap-swp-files.html
350 https://en.wikipedia.org/wiki/.DS_Store
351 https://en.wikipedia.org/wiki/Windows_thumbnail_cache#Thumbs.db

15.2 DataLad extensions

DataLad's commands cover a broad range of domain-agnostic use cases. However, there are extension packages that can add specialized functionality with additional commands. Table 15.1 lists a number of such extensions.

DataLad extensions are shipped as separate Python packages, and are *not* included in DataLad itself. Instead, users needing a particular extension can install the extension package – either on top of DataLad, if already installed, or on its own. In the latter case, the extension will then pull in DataLad core automatically, with no need to first or simultaneously install DataLad itself explicitly. The installation is done with standard Python package managers, such as PIP, and beyond installation of the package, no additional setup is required.

DataLad extensions listed here are of various maturity levels. Check out their documentation and the sections or chapters associated with an extension to find out more about them.

Table 15.1: Selection of available DataLad extensions. A more up-to-date list can be found on PyPi[352]

Name	Description
container[353]	Equips DataLad's `run`/`rerun` functionality with the ability to transparently execute commands in containerized computational environments. The section *Computational reproducibility with software containers* (page 166) demonstrates how this extension can be used, as well as the usecase *An automatically and computationally reproducible neuroimaging analysis from scratch* (page 420).
crawler[354]	One of the initial goals behind DataLad was to provide access to already existing data resources. With `crawl-init`/`crawl` commands, this extension allows to automate creation of DataLad datasets from resources available online, and efficiently keep them up-to-date. The majority of datasets in THE DATALAD SUPERDATASET /// on datasets.datalad.org[355] are created and updated using this extension functionality.
hirni[356]	A neuroimaging specific extension to allow reproducible DICOM to BIDS conversion of (f)MRI data. The chapter . . . introduces this extension.
htcondor[357]	Enhances DataLad with the ability for remote execution via the job scheduler HTCondor[358].
metalad[359]	Equips DataLad with an alternative command suite and advanced tooling for metadata handling (extraction, aggregation, reporting).
mihextras[360]	Special-interest commands and previews for future DataLad additions.
neuroimaging[361]	Metadata extraction support for a range of standards common to neuroimaging data. The usecase *An automatically and computationally reproducible neuroimaging analysis from scratch* (page 420) demonstrates how this extension can be used.
osf[362]	Enables DataLad to interface and work with the Open Science Framework[363]. Use it to publish your dataset's data to an OSF project, thus utilizing the OSF for dataset storage and sharing.
rclone-remote[364]	Enables DataLad to push and pull to all third party providers with no native Git support that are supported by rclone[365].
ukbiobank[366]	Equips DataLad with a set of commands to obtain and monitor imaging data releases of the UKBiobank[367]. An introduction can be found in chapter
xnat[368]	Equips DataLad with a set of commands to track XNAT[369] projects. An alternative, more basic method to retrieve data from an XNAT server is outlined in section *Configure custom data access* (page 290).

To install a DataLad extension, use

```
$ pip install <extension-name>
```

such as in

```
$ pip install datalad-container
```

Afterwards, the new DataLad functionality the extension provides is readily available.

Some extensions could also be available from the software distribution (e.g., NeuroDebian or conda) you used to install DataLad itself. Visit github.com/datalad/datalad-extensions/[371] to review available versions and their status.

15.3 DataLad's result hooks

If you are particularly keen on automating tasks in your datasets, you may be interested in running DataLad commands automatically as soon as previous commands are executed and resulted in particular outcomes or states. For example, you may want to automatically **unlock** all dataset contents right after an installation in one go. However, you'd also want to make sure that the **install** command was *successful* before attempting an **unlock**. Therefore, you would like to automatically run the **datalad unlock .** command right after the **datalad install** command, *but only* if the previous **install** command was successful.

Such automation allows for flexible and yet automatic responses to the results of DataLad commands, and can be done with DataLad's *result hooks*. Generally speaking, hooks[372] intercept function calls or events and allow to extend the functionality of a program. DataLad's result hooks are calls to other DataLad commands after the command resulted in a specified result – such as a successful install.

To understand how hooks can be used and defined, we have to briefly mention DataLad's *command result evaluations*. Whenever a DataLad command is executed, an internal evaluation generates a *report* on the status and result of the command. To get a glimpse into such an evaluation, you can call any DataLad command with the datalad option -f/--output-format

---

[352] https://pypi.org/search/?q=datalad
[353] http://docs.datalad.org/projects/container
[354] http://docs.datalad.org/projects/crawler
[355] http://datasets.datalad.org/
[356] http://docs.datalad.org/projects/hirni
[357] https://github.com/datalad/datalad-htcondor
[358] https://research.cs.wisc.edu/htcondor/
[359] http://docs.datalad.org/projects/metalad
[360] https://datalad-mihextras.readthedocs.io
[361] https://datalad-neuroimaging.readthedocs.io
[362] http://docs.datalad.org/projects/osf
[363] https://osf.io/
[364] https://github.com/datalad/git-remote-rclone
[365] https://rclone.org/
[366] https://github.com/datalad/datalad-ukbiobank
[367] https://www.ukbiobank.ac.uk//
[368] https://github.com/datalad/datalad-xnat
[369] https://www.xnat.org/
[371] https://github.com/datalad/datalad-extensions/
[372] https://en.wikipedia.org/wiki/Hooking

`<default, json, json_pp, tailored, '<template>'>` to return the command result evaluations with a specific formatting. Here is how this can look like for a **datalad create**:

```
$ datalad -f json_pp create somedataset
 [INFO   ] Creating a new annex repo at /tmp/somedataset
 {
   "action": "create",
   "path": "/tmp/somedataset",
   "refds": null,
   "status": "ok",
   "type": "dataset"
 }
```

Internally, this is useful for final result rendering, error detection, and logging. However, by using hooks, you can utilize these evaluations for your own purposes and "hook" in more commands whenever an evaluation fulfills your criteria.

To be able to specify matching criteria, you need to be aware of the potential criteria you can match against. The evaluation report is a dictionary with key:value pairs. Table 15.2 provides an overview on some of the available keys and their possible values.

Table 15.2: Common result keys and their values. This is only a selection of available key-value pairs. The actual set of possible key-value pairs is potentially unlimited, as any third-party extension could introduce new keys, for example. If in doubt, use the `-f`/`--output-format` option with the command of your choice to explore how your matching criteria may look like.

Key name	Values
`action`	`get`, `install`, `drop`, `status`, ... (any command's name)
`type`	`file`, `dataset`, `symlink`, `directory`
`status`	`ok`, `notneeded`, `impossible`, `error`
`path`	The path the previous command operated on

These key-value pairs provide the basis to define matching rules that – once met – can trigger the execution of custom hooks. To define a hook based on certain command results, two configuration variables need to be set:

`datalad.result-hook.<name>.match-json`

and

`datalad.result-hook.<name>.call-json`

Here is what you need to know about these variables:

- The `<name>` part of the configurations is the same for both variables, and can be an arbitrarily[373] chosen name that serves as an identifier for the hook you are defining.

- The first configuration variable, `datalad.result-hook.<name>.match-json`, defines the requirements that a result evaluation needs to match in order to trigger the hook.

- The second configuration variable, `datalad.result-hook.<name>.call-json`, defines what the hook execution comprises. It can be any DataLad command of your choice.

And here is how to set the values for these variables:

---

[373] It only needs to be compatible with **git config**. This means that it for example should not contain any dots (.).

- When set via the **git config** command, the value for datalad.result-hook.<name>.
  match-json needs to be specified as a JSON-encoded dictionary with any number of keys,
  such as

  {"type": "file", "action": "get", "status": "notneeded"}

  This translates to: "Match a "not-needed" after **datalad get** of a file." If all specified values
  in the keys in this dictionary match the values of the same keys in the result evaluation,
  the hook is executed. Apart from == evaluations, in, not in, and != are supported. To
  make use of such operations, the test value needs to be wrapped into a list, with the first
  item being the operation, and the second value the test value, such as

  {"type": ["in", ["file", "directory"]], "action": "get", "status": "notneeded"}

  This translates to: "Match a "not-needed" after **datalad get** of a file or directory." Another
  example is

  {"type":"dataset","action":"install","status":["eq", "ok"]}

  which translates to: "Match a successful installation of a dataset".

- The value for datalad.result-hook.<name>.call-json is specified in its Python notation,
  and its options – when set via the **git config** command – are specified as a JSON-encoded
  dictionary with keyword arguments. Conveniently, a number of string substitutions are
  supported: a dsarg argument expands to the dataset given to the initial command the
  hook operates on, and any key from the result evaluation can be expanded to the respec-
  tive value in the result dictionary. Curly braces need to be escaped by doubling them. This
  is not the easiest specification there is, but its also not as hard as it may sound. Here is
  how this could look like for a **datalad unlock**:

  $ unlock {{"dataset": "{dsarg}", "path": "{path}"}}

  This translates to "unlock the path the previous command operated on, in the dataset the
  previous command operated on". Another example is this run command:

  $ run {{"cmd": "cp ~/Templates/standard-readme.txt {path}/README", "dataset": "
  →{dsarg}", "explicit": true}}

  This translate to "execute a run command in the dataset the previous command op-
  erated on. In this run command, copy a README template file from ~/Templates/
  standard-readme.txt and place it into the newly created dataset." A final example is
  this:

  $ run_procedure {{"dataset":"{path}","spec":"cfg_metadatatypes bids"}}

  This hook will run the procedure cfg_metadatatypes with the argument bids and thus
  set the standard metadata extractor to be bids.

As these variables are configuration variables, they can be set via **git config** – either for the
dataset (--local), or the user (--global)[374]:

---

[374] To re-read about the **git config** command and other configurations of DataLad and its underlying tools, go
back to the chapter on Configurations, starting with *DIY configurations* (page 112). **Note that hooks are only
read from Git's config files, not .datalad/config!** Else, this would pose a severe security risk, as it would allow
installed datasets to alter DataLad commands to perform arbitrary executions on a system.

```
$ git config --global --add datalad.result-hook.readme.call-json 'run {{"cmd":"cp ~/
↪Templates/standard-readme.txt {path}/README", "outputs":["{path}/README"], "dataset":"
↪{path}","explicit":true}}'
$ git config --global --add datalad.result-hook.readme.match-json '{"type": "dataset",
↪"action":"create","status":"ok"}'
```

Here is what this writes to the `~/.gitconfig` file:

```
[datalad "result-hook.readme"]
    call-json = run {{\"cmd\":\"cp ~/Templates/standard-readme.txt {path}/README\", \
↪"outputs\":[\"{path}/READ>
    match-json = {\"type\": \"dataset\",\"action\":\"create\",\"status\":\"ok\"}
```

Note how characters such as quotation marks are automatically escaped via backslashes. If you want to set the variables "by hand" with an editor instead of using **git config**, pay close attention to escape them as well.

Given this configuration in the global `~/.gitconfig` file, the "readme" hook would be executed whenever you successfully create a new dataset with **datalad create**. The "readme" hook would then automatically copy a file, `~/Templates/standard-readme.txt` (this could be a standard README template you defined), into the new dataset.

15.4 Configure custom data access

DataLad can download files via the `http`, `https`, `ftp`, and `s3` protocol from various data storage solutions via its downloading commands (**datalad download-url**, **datalad addurls**, **datalad get**). If data retrieval from a storage solution requires *authentication*, for example via a username and password combination, DataLad provides an interface to query, request, and store the most common type of credentials that are necessary to authenticate, for a range of authentication types. There are a number of natively supported types of authentication and out-of-the box access to a broad range of access providers, from common solutions such as S3[375] to special purpose solutions, such as LORIS[376]. However, beyond natively supported services, custom data access can be configured as long as the required authentication and credential type are supported. This makes DataLad even more flexible for retrieving data.

Basic process

For any supported access type that requires authentication, the procedure is always the same: Upon first access via any downloading command, users will be prompted for their credentials from the command line. Subsequent downloads handle authentication in the background as long as the credentials stay valid. An example of this credential management is shown in the usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433): Data is stored in S3 buckets that require authentication with AWS credentials. The first **datalad get** to retrieve any of the data will prompt for the credentials from the terminal. If the given credentials are valid, the requested data will be downloaded, and all subsequent retrievals via **get** will authenticate automatically, without user input, as long as the entered credentials stay valid.

---

[375] https://aws.amazon.com/s3/?nc1=h_ls
[376] https://loris.ca/

> 💡 **M15.3 How does the authentication work?**
>
> Passwords, user names, tokens, or any other login information is stored in your system's (encrypted) keyring[377]. It is a built-in credential store, used in all major operating systems, and can store credentials securely. DataLad uses the Python keyring[378] package to access the keyring. In addition to a standard interface to the keyring, this library also has useful special purpose backends that come in handy in corner cases such as HPC/cluster computing, where no interactive sessions are available.
>
> ---
> [377] https://en.wikipedia.org/wiki/GNOME_Keyring
> [378] https://keyring.readthedocs.io/en/latest/

If a particular storage solution requires authentication but it is not known to DataLad yet, the download will fail. Here is how this looks like if data is retrieved from a server that requires HTTP authentication, but DataLad – or the dataset – lacks a configuration for data access about this server:

```
$ datalad download-url  \
  https://example.com/myuser/protected/path/to/file
  [INFO   ] Downloading 'https://example.com/myuser/protected/path/to/file' into 'local/
↪path/'
  Authenticated access to https://example.com/myuser/protected/path/to/file has failed.
  Would you like to setup a new provider configuration to access url? (choices: [yes],␣
↪no): yes
```

However, data access can be configured by the user if the required authentication and credential type are supported by DataLad (a list is given in the hidden section below). With a data access configuration in place, commands such as **datalad download-url** or **datalad addurls** can work with urls the point to the location of the data to be retrieved, and **datalad get** is enabled to retrieve file contents from these sources.

The configuration can either be done in the terminal upon a prompt from the command line when a download fails due to a missing provider configuration as shown above, or by placing a configuration file for the required data access into `.datalad/providers/<provider-name>.cfg`. The following information is needed:

- An arbitrary name that the data access is identified with,

- a regular expression that can match a url one would want to download from,

- an authentication type, and

- a credential type.

The example below sheds some light one this.

> 💡 **M15.4 Which authentication and credential types are possible?**
>
> When configuring custom data access, credential and authentication type are required information. Below, we list the most common choices for these fields.
> Among the most common credential types, `'user_password'`, `'aws-s3'`, and `'token'` authentication is supported. For a full list, including some less common authentication types, please see the technical documentation of DataLad.
> For authentication, the most common supported solutions are `'html_form'`, `'http_auth'` (http and html form-based authentication[379]), `'http_basic_auth'` (http basic access[380]), `'http_digest_auth'` (digest access authentication[381]), `'bearer_token'` (http bearer to-

💡 ken authentication[382]) and `'aws-s3'`. A full list can be found in the technical docs.

---

[379] https://en.wikipedia.org/wiki/HTTP%2BHTML_form-based_authentication
[380] https://en.wikipedia.org/wiki/Basic_access_authentication
[381] https://en.wikipedia.org/wiki/Digest_access_authentication
[382] https://tools.ietf.org/html/rfc6750

Example: Data access to a server that requires basic HTTP authentication

Consider a private Apache web server[383] with an `.htaccess` file that configures a range of allowed users to access a certain protected directory on this server via basic HTTP authentication[384]. If opened in a browser, such a setup would prompt visitors of this directory on the web server for their username and password, and only grant access if valid credentials are entered. Unauthenticated requests cause `401 Unauthorized Status` responses.

By default, when DataLad attempts to retrieve files from this protected directory, the authentication and credential type that are required are unknown to DataLad and authentication fails. An attempt to download or get a file from this directory with DataLad can only succeed if a "provider configuration", i.e., a configuration how to access the data, for this specific web server with information on how to authenticate exists.

"Provider configurations" are small text files that either exist on a per-dataset level in `.datalad/providers/<name>.cfg`, or on a user-level in `~/.config/datalad/providers/<name>.cfg`. They can be created and saved by hand, or configured "on the fly" from the command line upon unsuccessful download attempts. A configuration file follows a similar structure as the example below:

```
[provider:my-webserver]
url_re = https://example.com/~myuser/protected/.*
credential = my-webserver
authentication_type = http_basic_auth

[credential:my-webserver]
type = user_password
```

For a *local*[385], i.e., dataset-specific, configuration, place the file into `.datalad/providers/my-webserver.cfg`. Subsequently, in the dataset that this file was placed into, downloading commands that point to `https://example.com/~myuser/protected/<path>` will ask (once) for the user's user name and password, and subsequently store these credentials. In order to make it a *global* configuration, i.e., enable downloads from the web server from within all datasets of the user, place the file into the users home directory under `~/.config/datalad/providers/my-webserver.cfg`.

If the file is generated "on the fly" from the terminal, it will prompt for exactly the same information as specified in the example above and write the required `.cfg` based on the given information. Note that this will configure data access *globally*, i.e., it will place the file under `~/.config/datalad/providers/<name>.cfg`. Here is how that would look like:

---

[383] https://httpd.apache.org/
[384] https://en.wikipedia.org/wiki/Basic_access_authentication
[385] To re-read on configurations and their scope, check out chapter *Tuning datasets to your needs* (page 112) again.

```
$ datalad download-url  https://example.com/~myuser/protected/my_protected_file
 [INFO   ] Downloading 'https://example.com/~myuser/protected/my_protected_file' into '/
↪tmp/ds/'
 Authenticated access to https://example.com/~myuser/protected/my_protected_file has␣
↪failed.
 Would you like to setup a new provider configuration to access url? (choices: [yes],␣
↪no): yes

 New provider name
 Unique name to identify 'provider' for https://example.com/~myuser/protected/my_
↪protected_file [https://example.com]:
 my-webserver

 New provider regular expression
 A (Python) regular expression to specify for which URLs this provider
 should be used [https://example\.com/\~myuser/protected/my_protected_file]:
 https://example.com/~myuser/protected/.*

 Authentication type
 What authentication type to use (choices: aws-s3, bearer_token, html_form,
 http_auth, http_basic_auth, http_digest_auth, loris-token, nda-s3, none, xnat):
 http_basic_auth

 Credential
 What type of credential should be used? (choices: aws-s3, loris-token, nda-s3,
 token, [user_password]):
 user_password

 Save provider configuration file
 Following configuration will be written to /home/me/.config/datalad/providers/my-
↪webserver.cfg:
 # Provider configuration file created to initially access
 # https://example.com/~myuser/protected/my_protected_file

 [provider:my-webserver]
 url_re = https://example.com/~myuser/protected/.*
 authentication_type = http_basic_auth
 # Note that you might need to specify additional fields specific to the
 # authenticator.  Fow now "look into the docs/source" of <class 'datalad.downloaders.
↪http.HTTPBasicAuthAuthenticator'>
 # http_basic_auth_
 credential = my-webserver

 [credential:my-webserver]
 # If known, specify URL or email to how/where to request credentials
 # url = ???
 type = user_password
  (choices: [yes], no):
 yes

 You need to authenticate with 'my-webserver' credentials.
 user: <user name>

 password: <password>
 password (repeat): <password>
 [INFO   ] http session: Authenticating into session for https://example.com/~myuser/
↪protected/my_protected_file
```

```
https://example.com/~myuser/protected/my_protected_file:   0%| | 0.00/611k
download_url(ok): /https://example.com/~myuser/protected/my_protected_file (file)
add(ok): my_protected_file (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
```

Subsequently, all downloads from `https://example.com/~myuser/protected/*` by the user will succeed. If something went wrong during this interactive configuration, delete or edit the file at `~/.config/datalad/providers/<name>.cfg`.

15.5 Remote Indexed Archives for dataset storage and backup

If DataLad datasets should be backed-up, made available for collaborations with others, or stored or managed in a central location, REMOTE INDEXED ARCHIVE (RIA) STOREs, dataset storage locations that allow for access to and collaboration on DataLad datasets, may be a suitable solution. They are flat, flexible, file-system based repository representations of any number of datasets, and they can exist on all standard computing infrastructure, be it personal computers, servers or compute clusters, or even super computing infrastructure – even on machines that do not have DataLad installed.

> ⚠️ **RIA availability**
>
> Setting up and interacting with RIA stores requires DataLad version `0.13.0` or higher. In order to understand this section, some knowledge on Git-internals and overcoming any fear of how checksums and UUIDs look can be helpful.

Technical details

RIA stores can be created or extended with a single command from within any dataset. DataLad datasets can subsequently be published into the datastore as a means of backing up a dataset or creating a dataset sibling to collaborate on with others. Alternatively, datasets can be cloned and updated from a RIA store just as from any other dataset location. The subsection *RIA store workflows* (page 298) a few paragraphs down will demonstrate RIA-store related functionality. But prior to introducing the user-facing commands, this section starts by explaining the layout and general concept of a RIA store.

Layout

RIA stores store DataLad datasets. Both the layout of the RIA store and the layout of the datasets in the RIA store are different from typical dataset layouts, though. If one were to take a look inside of a RIA store as it is set up by default, one would see a directory that contains a flat subdirectory tree with datasets represented as BARE GIT REPOSITORIES and an annex. Usually, looking inside of RIA stores is not necessary for RIA-related workflows, but it can help to grasp the concept of these stores.

The first level of subdirectories in this RIA store tree consists of the first three characters of the DATASET IDs of the datasets that lie in the store, and the second level of subdatasets contains the remaining characters of the dataset IDs. Thus, the first two levels of subdirectories in the tree are split dataset IDs of the datasets that are stored in them[398]. The code block below illustrates how a single DataLad dataset looks like in a RIA store, and the dataset ID of the dataset (946e8cac-432b-11ea-aac8-f0d5bf7b5561) is highlighted:

```
/path/to/my_riastore
├── 946
│   └── e8cac-432b-11ea-aac8-f0d5bf7b5561
│       ├── annex
│       │   └── objects
│       │       ├── 6q
│       │       │   └── mZ
│       │       │       └── MD5E-s93567133--7c93fc5d0b5f197ae8a02e5a89954bc8.nii.gz
│       │       │           └── MD5E-s93567133--7c93fc5d0b5f197ae8a02e5a89954bc8.nii.gz
│       │       ├── 6v
│       │       │   └── zK
│       │       │       └── MD5E-s2043924480--47718be3b53037499a325cf1d402b2be.nii.gz
│       │       │           └── MD5E-s2043924480--47718be3b53037499a325cf1d402b2be.nii.gz
│       │       ├── [...]
│       │       └── [...]
│       ├── archives
│       │   └── archive.7z
│       ├── branches
│       ├── config
│       ├── description
│       ├── HEAD
│       ├── hooks
│       │   ├── applypatch-msg.sample
│       │   ├── [...]
│       │   └── update.sample
│       ├── info
│       │   └── exclude
│       ├── objects
│       │   ├── 05
│       │   │   └── 3d25959223e8173497fa7f747442b72c31671c
│       │   ├── 0b
│       │   │   └── 8d0edbf8b042998dfeb185fa2236d25dd80cf9
│       │   ├── [...]
│       │   │   └── [...]
│       │   ├── info
│       │   └── pack
│       ├── refs
│       │   ├── heads
│       │   │   ├── git-annex
│       │   │   └── master
│       │   └── tags
│       ├── ria-layout-version
│       └── ria-remote-ebce196a-b057-4c96-81dc-7656ea876234
│           └── transfer
├── error_logs
└── ria-layout-version
```

If a second dataset gets published to the RIA store, it will be represented in a similar tree

---

[398] The two-level structure (3 ID characters as one subdirectory, the remaining ID characters as the next subdirectory) exists to avoid exhausting file system limits on the number of files/folders within a directory.

---

structure underneath its individual dataset ID. If *subdatasets* of a dataset are published into a RIA store, they are not represented *underneath* their superdataset, but are stored on the same hierarchy level as any other dataset. Thus, the dataset representation in a RIA store is completely flat[399]. With this hierarchy-free setup, the location of a particular dataset in the RIA store is only dependent on its DATASET ID. As the dataset ID is universally unique, gets assigned to a dataset at the time of creation, and does not change across the life time of a dataset, no two different datasets could have the same location in a RIA store.

The directory underneath the two dataset-ID-based subdirectories contains a *bare git repository* (highlighted above as well) that is a CLONE of the dataset.

> ### M15.5 What is a bare Git repository?
>
> A bare Git repository is a repository that contains the contents of the `.git` directory of regular DataLad datasets or Git repositories, but no worktree or checkout. This has advantages: The repository is leaner, it is easier for administrators to perform garbage collections, and it is required if you want to push to it at all times. You can find out more on what bare repositories are and how to use them here[386].
>
> Note that bare Git repositories can be cloned, and the clone of a bare Git repository will have a checkout and a worktree, thus resuming the shape that you are familiar with.
>
> ---
> [386] https://git-scm.com/book/en/v2/Git-on-the-Server-Getting-Git-on-a-Server

Inside of the bare GIT repository, the `annex` directory – just as in any standard dataset or repository – contains the dataset's keystore (object tree) under `annex/objects`[401]. In conjunction, keystore and bare Git repository are the original dataset – just differently represented, with no *working tree*, i.e., directory hierarchy that exists in the original dataset, and without the name it was created under, but stored under its dataset ID instead.

If necessary, the keystores (annex) can be (compressed) 7zipped[387] archives (`archives/`), either for compression gains, or for use on HPC-systems with inode[388] limitations[403]. Despite being 7zipped, those archives can be indexed and support relatively fast random read access. Thus, the entire key store can be put into an archive, re-using the exact same directory structure, and remains fully accessible while only using a handful of inodes, regardless of file number and size. If the dataset contains only annexed files, a complete dataset can be represented in about 25 inodes.

Taking all of the above information together, on an infrastructural level, a RIA store is fully self-contained, and is a plain file system storage, not a database. Everything inside of a RIA store is either a file, a directory, or a zipped archive. It can thus be set up on any infrastructure that

---

[399] Beyond datasets, the RIA store only contains the directory `error_logs` for error logging and the file `ria-layout-version` for a specification of the dataset tree layout in the store (last two lines in the code block above). The `ria-layout-version` is important because it identifies whether the keystore uses git-annex's `hashdirlower` (git-annex's default for bare repositories) or `hashdirmixed` layout (which is necessary to allow symlinked annexes, relevant for EPHEMERAL CLONEs). To read more about hashing in the key store, take a look at the docs[400].

[400] https://git-annex.branchable.com/internals/hashing/

[401] To re-read about how git-annex's object tree works, check out section *Data integrity* (page 85), and pay close attention to the hidden section. Additionally, you can find a lot of background information in git-annex's documentation[402].

[402] https://git-annex.branchable.com/internals/

[387] https://www.7-zip.org/

[388] https://en.wikipedia.org/wiki/Inode

[403] The usecase

shows how this feature can come in handy.

---

has a file system with directory and file representation, and has barely any additional software requirements (see below). Access to datasets in the store can be managed by using file system PERMISSIONS. With these attributes, a RIA store is a suitable solution for a number of usecases (back-up, single or multi-user dataset storage, central point for collaborative workflows, . . .), be that on private workstations, web servers, compute clusters, or other IT infrastructure.

> **M15.6 Software Requirements**
>
> On the RIA store hosting infrastructure, only 7z is to be installed, if the archive feature is desired. Specifically, no GIT, no GIT-ANNEX, and no otherwise running daemons are necessary. If the RIA store is set up remotely, the server needs to be SSH-accessible.
> On the client side, you need DataLad version 0.13.0 or later. Starting with this version, DataLad has the `create-sibling-ria` command and the git-annex ora-remote special remote that is required to get annexed dataset contents into a RIA store.

git-annex ORA-remote special remotes

On a technical level, beyond being a directory tree of datasets, a RIA store is by default a GIT-ANNEX ORA-remote (optional remote access) special remote of a dataset. This allows to not only store the history of a dataset, but also all annexed contents.

> **M15.7 What is a special remote?**
>
> A special-remote[389] is an extension to Git's concept of remotes, and can enable git-annex to transfer data to and from places that are not Git repositories (e.g., cloud services or external machines such as an HPC system). Don't envision a special-remote as a physical place or location – a special-remote is just a protocol that defines the underlying *transport* of your files *to* and *from* a specific location.
>
> ---
> [389] https://git-annex.branchable.com/special_remotes/

The git-annex ora-remote special remote is referred to as a "storage sibling" of the original dataset. It is similar to git-annex's built-in directory[390] special remote (but works remotely and uses the `hashdir_mixed`[399] keystore layout). Thanks to the git-annex ora-remote, RIA stores can have regular git-annex key storage and retrieval of keys from (compressed) 7z archives in the RIA store works. Put simple, annexed contents of datasets can only be pushed into RIA stores if they have a git-annex ora-remote.

Certain applications will not require special remote features. The usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433) shows an example where git-annex key storage is explicitly not wanted. Other applications may require *only* the special remote, such as cases where Git isn't installed on the RIA store hosting infrastructure. For most storage or back-up scenarios, special remote capabilities are useful, though, and thus the default.

By default, the `datalad create-sibling-ria` command will automatically create a dataset representation in a RIA store (and set up the RIA store, if it does not exist), and configure a sibling to allow publishing to the RIA store and updating from it. With special remote capabilities enabled, the command will automatically create the special remote as a `storage-sibling` and link it to the RIA-sibling. With the sibling and special remote set up, upon an invocation of

---
[390] https://git-annex.branchable.com/special_remotes/directory/

`datalad push --to <sibling>`, the complete dataset contents, including annexed contents, will be published to the RIA store, with no further setup or configuration required[404].

To disable the storage sibling completely, invoke `datalad create-sibling-ria` with the argument `--storage-sibling=off`. Note that DataLad versions <0.14 need to use the flag `--no-storage-sibling`, which is deprecated starting with DataLad 0.14.0. To create a RIA store with *only* special remote storage, starting from DataLad version 0.14.0 you can invoke `datalad create-sibling-ria` with the argument `--storage-sibling=only`.

Advantages of RIA stores

Storing datasets in RIA stores has a number of advantages that align well with the demands of central dataset management on shared compute infrastructure, but are also well suited for most back-up and storage applications. In a RIA store layout, the first two levels of subdirectories can host any number of keystores and bare repositories. As datasets are identified via ID and stored *next to each other* underneath the top-level RIA store directory, the store is completely flexible and extendable, and regardless of the number or nature of datasets inside of the store, a RIA store keeps a homogeneous directory structure. This aids the handling of large numbers of repositories, because unique locations are derived from *dataset/repository properties* (their ID) rather than a dataset name or a location in a complex dataset hierarchy. Because the dataset representation in the RIA store is a bare repository, "house-keeping" as well as query tasks can be automated or performed by data management personnel with no domain-specific knowledge about dataset contents. Short maintenance scripts can be used to automate basically any task that is of interest and possible in a dataset, but across the full RIA store. A few examples are:

- Copy or move annex objects into a 7z archive.

- Find dataset dependencies across all stored datasets by returning the dataset IDs of sub-datasets recorded in each dataset.

- Automatically return the number of commits in each repository.

- Automatically return the author and time of the last dataset update.

- Find all datasets associated with specific authors.

- Clean up unnecessary files and minimize a (or all) repository with GIT's garbage collection (gc)[391] command.

The usecase *Building a scalable data storage for scientific computing* (page 443) demonstrates the advantages of this in a large scientific institute with central data management. Due to the git-annex ora-remote special remote, datasets can be exported and stored as archives to save disk space.

RIA store workflows

The user facing commands for interactions with a RIA store are barely different from standard DataLad workflows. The paragraphs below detail how to create and populate a RIA store, how to clone datasets and retrieve data from it, and also how to handle permissions or hide technicalities.

---

[404] To re-read about publication dependencies and why this is relevant to annexed contents in the dataset, checkout section *Beyond shared infrastructure* (page 177).

[391] https://git-scm.com/docs/git-gc

Creating or publishing to RIA stores

A dataset can be added into an existing or not yet existing RIA store by running the **datalad create-sibling-ria** command (datalad-create-sibling-ria manual), and subsequently published into the store using **datalad push**. Just like the **datalad siblings add** command, for **datalad create-sibling-ria**, an arbitrary sibling name (with the -s/--name option) and a URL to the location of the store (as a positional argument) need to be specified. In the case of RIA stores, the URL takes the form of a ria+ URL, and the looks of this URL are dependent on where the RIA store (should) exists, or rather, which file transfer protocol (SSH or file) is used:

- A URL to an SSH-accessible server has a ria+ssh:// prefix, followed by user and hostname specification and an **absolute** path: ria+ssh://[user@]hostname:/absolute/path/to/ria-store

- A URL to a store on a local file system has a ria+file:// prefix, followed by an **absolute** path: ria+file:///absolute/path/to/ria-store

Note that it is always required to specify an ABSOLUTE PATH in the URL!

> ⚠️ **If you code along, make sure to check the next findoutmore!**
>
> The upcoming demonstration of RIA stores uses the DataLad-101 dataset the was created throughout the Basics of this handbook. If you want to execute these code snippets on a DataLad-101 dataset you created, the modification described in the findoutmore below needs to be done first.

> 💡 **M15.8 If necessary, adjust the submodule path!**
>
> Back in *Subdataset publishing* (page 212), in order to appropriately reference and link subdatasets on hostings sites such as GITHUB, we adjusted the submodule path of the subdataset in .gitmodules to point to a published subdataset on GitHub:
>
> ```
> # in DataLad-101
> $ cat .gitmodules
> [submodule "recordings/longnow"]
>         path = recordings/longnow
>         url = https://github.com/datalad-datasets/longnow-podcasts.git
>         datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e
>         datalad-url = https://github.com/datalad-datasets/longnow-podcasts.git
> [submodule "midterm_project"]
>         path = midterm_project
>         url = https://github.com/adswa/midtermproject
>         datalad-id = fea0e9c7-7932-41d5-8d57-97ac9ac6755a
> ```
>
> Later in this demonstration we would like to publish the subdataset to a RIA store and retrieve it automatically from this store – retrieval is only attempted from a store, however, if no other working source is known. Therefore, we will remove the reference to the published dataset prior to this demonstration and replace it with the path it was originally referenced under.
>
> ```
> # in DataLad-101
> $ datalad subdatasets --contains midterm_project --set-property url ./midterm_
> ↪project
> subdataset(ok): midterm_project (dataset)
> ```

To demonstrate the basic process, we will create a RIA store on a local file system to publish the
DataLad-101 dataset from the handbook's "Basics" section to. In the example below, the RIA
sibling gets the name ria-backup. The URL uses the file protocol and points with an absolute
path to the not yet existing directory myriastore.

```
# inside of the dataset DataLad-101
$ datalad create-sibling-ria -s ria-backup ria+file:///home/me/myriastore
[INFO] Start checking pre-existing sibling configuration Dataset(/home/me/dl-101/DataLad-
→101)
[INFO] Discovered sibling here in dataset at /home/me/dl-101/DataLad-101
[INFO] Discovered sibling roommate in dataset at /home/me/dl-101/DataLad-101
[INFO] Discovered sibling gin in dataset at /home/me/dl-101/DataLad-101
[INFO] Finished checking pre-existing sibling configuration Dataset(/home/me/dl-101/
→DataLad-101)
[INFO] create siblings 'ria-backup' and 'ria-backup-storage' ...
[INFO] Fetching updates for Dataset(/home/me/dl-101/DataLad-101)
[INFO] Configure additional publication dependency on "ria-backup-storage"
create-sibling-ria(ok): /home/me/dl-101/DataLad-101 (dataset)
```

Afterwards, the dataset has two additional siblings: ria-backup, and ria-backup-storage.

```
$ datalad siblings
.: here(+) [git]
.: roommate(+) [../mock_user/DataLad-101 (git)]
.: gin(+) [/home/me/pushes/DataLad-101 (git)]
.: ria-backup-storage(+) [ora]
.: ria-backup(-) [/home/me/myriastore/8e0/4afb0-af85-4070-be29-858d30d85017 (git)]
```

The storage sibling is the git-annex ora-remote and is set up automatically – unless
create-sibling-ria is run with --storage-sibling=off (in DataLad versions >0.14.) or
--no-storage-sibling (in versions <0.14). By default, it has the name of the RIA sibling,
suffixed with -storage, but alternative names can be supplied with the --storage-name option.

> **M15.9 Take a look into the store**
>
> Right after running this command, a RIA store has been created in the specified location:

```
$ tree /home/me/myriastore
/home/me/myriastore
├── 8e0
│   └── 4afb0-af85-4070-be29-858d30d85017
│       ├── annex
│       │   └── objects
│       ├── archives
│       ├── branches
│       ├── config
│       ├── config.dataladlock
│       ├── description
│       ├── HEAD
│       ├── hooks
│       │   ├── applypatch-msg.sample
│       │   ├── commit-msg.sample
│       │   ├── fsmonitor-watchman.sample
│       │   ├── post-update.sample
│       │   ├── pre-applypatch.sample
│       │   ├── pre-commit.sample
│       │   ├── pre-merge-commit.sample
│       │   ├── prepare-commit-msg.sample
│       │   ├── pre-push.sample
│       │   ├── pre-rebase.sample
│       │   ├── pre-receive.sample
│       │   ├── push-to-checkout.sample
│       │   └── update.sample
│       ├── info
│       │   └── exclude
│       ├── objects
│       │   ├── info
│       │   └── pack
│       ├── refs
│       │   ├── heads
│       │   └── tags
│       └── ria-layout-version
├── error_logs
└── ria-layout-version

15 directories, 20 files
```

Note that there is one dataset represented in the RIA store. The two-directory structure
it is represented under corresponds to the dataset ID of `DataLad-101`:

```
# The dataset ID is stored in .datalad/config
$ cat .datalad/config
[datalad "dataset"]
        id = 8e04afb0-af85-4070-be29-858d30d85017
```

In order to publish the dataset's history and all its contents into the RIA store, a single **datalad
push** to the RIA sibling suffices:

```
$ datalad push --to ria-backup
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Determine push target
[INFO] Push refspecs
```

(continues on next page)

```
[INFO] Transfer data
copy(ok): books/TLCL.pdf (file) [to ria-backup-storage...]
copy(ok): books/bash_guide.pdf (file) [to ria-backup-storage...]
copy(ok): books/byte-of-python.pdf (file) [to ria-backup-storage...]
copy(ok): books/progit.pdf (file) [to ria-backup-storage...]
[INFO] Transfer data
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
[INFO] Start resolving deltas
publish(ok): . (dataset) [refs/heads/master->ria-backup:refs/heads/master [new branch]]
publish(ok): . (dataset) [refs/heads/git-annex->ria-backup:refs/heads/git-annex [new␣
↪branch]]
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101)
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101)
```

> 💡 **M15.10 Take another look into the store**
>
> Now that dataset contents have been pushed to the RIA store, the bare repository contains them, although their representation is not human-readable. But worry not – this representation only exists in the RIA store. When cloning this dataset from the RIA store, the clone will be in its standard human-readable format.

```
$ tree /home/me/myriastore
/home/me/myriastore
├── 8e0
│   └── 4afb0-af85-4070-be29-858d30d85017
│       ├── annex
│       │   └── objects
│       │       ├── G6
│       │       │   └── Gj
│       │       │       └── MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf
│       │       │           └── MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf
│       │       ├── jf
│       │       │   └── 3M
│       │       │       └── MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
│       │       │           └── MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
│       │       ├── WF
│       │       │   └── Gq
│       │       │       └── MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
│       │       │           └── MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
│       │       └── z1
│       │           └── Q8
│       │               └── MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
│       │                   └── MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
│       ├── archives
│       ├── branches
│       ├── config
│       │   ├── pre-push.sample
│       │   ├── pre-rebase.sample
│       │   ├── pre-receive.sample
│       │   ├── push-to-checkout.sample
│       │   └── update.sample
│       ├── info
│       │   └── exclude
│       ├── objects
│       │   ├── info
│       │   └── pack
│       │       ├── pack-f90d1d8e11495d22cc738734c458771c99287a61.idx
│       │       └── pack-f90d1d8e11495d22cc738734c458771c99287a61.pack
│       ├── ora-remote-7f049ef9-351d-4df5-974a-63e992128c31
│       │   └── transfer
│       ├── refs
│       │   ├── heads
│       │   │   ├── git-annex
│       │   │   └── master
│       │   └── tags
│       └── ria-layout-version
├── error_logs
└── ria-layout-version

29 directories, 28 files
```

A second dataset can be added and published to the store in the very same way. As a demonstration, we'll do it for the `midterm_project` subdataset:

```
$ cd midterm_project
$ datalad create-sibling-ria -s ria-backup ria+file:///home/me/myriastore
```

```
[INFO] Start checking pre-existing sibling configuration Dataset(/home/me/dl-101/DataLad-
↪101/midterm_project)
[INFO] Discovered sibling here in dataset at /home/me/dl-101/DataLad-101/midterm_project
[INFO] Discovered sibling datalad in dataset at /home/me/dl-101/DataLad-101/midterm_
↪project
[INFO] Discovered sibling github in dataset at /home/me/dl-101/DataLad-101/midterm_
↪project
[INFO] Finished checking pre-existing sibling configuration Dataset(/home/me/dl-101/
↪DataLad-101/midterm_project)
[INFO] create siblings 'ria-backup' and 'ria-backup-storage' ...
[INFO] Fetching updates for Dataset(/home/me/dl-101/DataLad-101/midterm_project)
[INFO] Configure additional publication dependency on "ria-backup-storage"
create-sibling-ria(ok): /home/me/dl-101/DataLad-101/midterm_project (dataset)


$ datalad push --to ria-backup
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Transfer data
copy(ok): .datalad/environments/midterm-software/image (file) [to ria-backup-storage...]
copy(ok): pairwise_relationships.png (file) [to ria-backup-storage...]
copy(ok): prediction_report.csv (file) [to ria-backup-storage...]
[INFO] Transfer data
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
[INFO] Start resolving deltas
publish(ok): . (dataset) [refs/heads/master->ria-backup:refs/heads/master [new branch]]
publish(ok): . (dataset) [refs/heads/git-annex->ria-backup:refs/heads/git-annex [new⏎
↪branch]]
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101/midterm_project)
[INFO] Finished push of Dataset(/home/me/dl-101/DataLad-101/midterm_project)
```

> 💡 **M15.11 Take a look into the RIA store after a second dataset has been added**
>
> With creating a RIA sibling to the RIA store and publishing the contents of the
> midterm_project subdataset to the store, a second dataset has been added to the data-
> store. Note how it is represented on the same hierarchy level as the previous dataset,
> underneath its dataset ID (note that the output is cut off for readability):
>
> ```
> $ cat .datalad/config
> [datalad "dataset"]
>         id = fea0e9c7-7932-41d5-8d57-97ac9ac6755a
> [datalad "containers.midterm-software"]
>         image = .datalad/environments/midterm-software/image
>         cmdexec = singularity exec {img} {cmd}
> ```

```
$ tree /home/me/myriastore
/home/me/myriastore
├── 8e0
│   └── 4afb0-af85-4070-be29-858d30d85017
│       ├── annex
│       │   └── objects
│       │       ├── G6
│       │       │   └── Gj
│       │       │       └── MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf
│       │       │           └── MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf
│       │       ├── jf
│       │       │   └── 3M
│       │       │       └── MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
│       │       │           └── MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
│       │       ├── WF
│       │       │   └── Gq
│       │       │       └── MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
│       │       │           └── MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
│       │       └── z1
│       │           └── Q8
│       │               └── MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
│       │                   └── MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
│       ├── archives
│       ├── branches
│       ├── config
│       │   ├── pre-push.sample
│       │   ├── pre-rebase.sample
│       │   ├── pre-receive.sample
│       │   ├── push-to-checkout.sample
│       │   └── update.sample
│       ├── info
│       │   └── exclude
│       ├── objects
│       │   ├── info
│       │   └── pack
│       │       ├── pack-f90d1d8e11495d22cc738734c458771c99287a61.idx
│       │       └── pack-f90d1d8e11495d22cc738734c458771c99287a61.pack
│       ├── ora-remote-7f049ef9-351d-4df5-974a-63e992128c31
│       │   └── transfer
│       ├── refs
│       │   ├── heads
│       │   │   ├── git-annex
│       │   │   └── master
│       │   └── tags
│       └── ria-layout-version
└── error_logs
```

Thus, in order to create and populate RIA stores, only the commands **datalad create-sibling-ria** and **datalad push** are required.

Cloning and updating from RIA stores

Cloning from RIA stores is done via **datalad clone** from a ria+ URL, suffixed with a dataset identifier. Depending on the protocol being used, the URLs are composed similarly to during

---

sibling creation:

- A URL to a RIA store on an SSH-accessible server takes the same format as before: `ria+ssh://[user@]hostname:/absolute/path/to/ria-store`

- A URL to a RIA store on a local file system also looks like during sibling creation: `ria+file:///absolute/path/to/ria-store`

- A URL for read (without annex) access to a store via HTTP (e.g., to a RIA store like store.datalad.org[392], through which the HCP dataset is published) looks like this: `ria+http://store.datalad.org:/absolute/path/to/ria-store`

The appropriate `ria+` URL needs to be suffixed with a # sign and a dataset identifier. One way this can be done is via the dataset ID. Here is how to clone the `DataLad-101` dataset from the RIA store using its dataset ID:

```
$ datalad clone ria+file:///home/me/myriastore#8e04afb0-af85-4070-be29-858d30d85017␣
↪myclone
[INFO] Cloning dataset to Dataset(/home/me/dl-101/myclone)
[INFO] Attempting to clone from file:///home/me/myriastore/8e0/4afb0-af85-4070-be29-
↪858d30d85017 to /home/me/dl-101/myclone
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/myclone)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Configure additional publication dependency on "ria-backup-storage"
configure-sibling(ok): . (sibling)
install(ok): /home/me/dl-101/myclone (dataset)
action summary:
  configure-sibling (ok: 1)
  install (ok: 1)
```

There are two downsides to this method: For one, it is hard to type, remember, and know the dataset ID of a desired dataset. Secondly, if no additional path is given to **datalad clone**, the resulting dataset clone would be named after its ID. An alternative, therefore, is to use an *alias* for the dataset. This is an alternative dataset identifier that a dataset in a RIA store can be configured with.

> **M15.12 Configure an alias for a dataset**
>
> In order to define an alias for an individual dataset in a store, one needs to create an `alias/` directory in the root of the datastore and place a SYMLINK of the desired name to the dataset inside of it. Here is how it is done, for the midterm project dataset:
> First, create an `alias/` directory in the store:
>
> ```
> $ mkdir /home/me/myriastore/alias
> ```
>
> Afterwards, place a SYMLINK with a name of your choice to the dataset inside of it. Here, we create a symlink called `midterm_project`:
>
> ```
> $ ln -s /home/me/myriastore/fea/0e9c7-7932-41d5-8d57-97ac9ac6755a /home/me/
> ↪myriastore/alias/midterm_project
> ```
>
> Here is how it looks like inside of this directory:

---

[392] http://store.datalad.org/

💡
```
$ tree /home/me/myriastore/alias
/home/me/myriastore/alias
└── midterm_project -> /home/me/myriastore/fea/0e9c7-7932-41d5-8d57-97ac9ac6755a

1 directory, 0 files
```

Afterwards, the alias name, prefixed with a ~, can be used as a dataset identifier:

```
datalad clone ria+file:///home/me/myriastore#~midterm_project
[INFO] Cloning dataset to Dataset(/home/me/dl-101/midterm_project)
[INFO] Attempting to clone from file:///home/me/myriastore/alias/midterm_project to␣
↪/home/me/dl-101/midterm_project
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/midterm_project)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Configure additional publication dependency on "ria-backup-storage"
configure-sibling(ok): . (sibling)
install(ok): /home/me/dl-101/midterm_project (dataset)
action summary:
  configure-sibling (ok: 1)
  install (ok: 1)
```

This makes it easier for others to clone the dataset and will provide a sensible default name for the clone if no additional path is provided in the command.

Note that it is even possible to create "aliases of an aliases" – symlinking an existing alias-symlink (in the example above midterm_project) under another name in the alias/ directory is no problem. This could be useful if the same dataset needs to be accessible via several aliases, or to safeguard against common spelling errors in alias names.

The dataset clone is just like any other dataset clone. Contents stored in GIT are present right after cloning, while the contents of annexed files is not yet retrieved from the store and can be obtained with a **datalad get**.

```
$ cd myclone
$ tree
.
├── books
│   ├── bash_guide.pdf -> ../.git/annex/objects/WF/Gq/MD5E-s1198170--
↪0ab2c121bcf68d7278af266f6a399c5f.pdf/MD5E-s1198170--0ab2c121bcf68d7278af266f6a399c5f.pdf
│   ├── byte-of-python.pdf -> ../.git/annex/objects/z1/Q8/MD5E-s4208954--
↪ab3a8c2f6b76b18b43c5949e0661e266.pdf/MD5E-s4208954--ab3a8c2f6b76b18b43c5949e0661e266.pdf
│   ├── progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-s12465653--
↪05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--05cd7ed561d108c9bcf96022bc78a92c.
↪pdf
│   └── TLCL.pdf -> ../.git/annex/objects/jf/3M/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf
├── code
│   ├── list_titles.sh
│   └── nested_repos.sh
├── Gitjoke2.txt
├── midterm_project
├── notes.txt
└── recordings
    ├── interval_logo_small.jpg
    ├── longnow
    ├── podcasts.tsv
    └── salt_logo_small.jpg
```

(continues on next page)

```
5 directories, 11 files
```

To demonstrate file retrieval from the store, let's get an annexed file:

```
$ datalad get books/progit.pdf
get(ok): books/progit.pdf (file) [from ria-backup-storage...]
```

> 💡 **M15.13 What about creating RIA stores and cloning from RIA stores with different protocols**
>
> Consider setting up and populating a RIA store on a server via the `file` protocol, but cloning a dataset from that store to a local computer via `SSH` protocol. Will this be a problem for file content retrieval? No, in all standard situations, DataLad will adapt to this. Upon cloning the dataset with a different URL than it was created under, enabling the special remote will initially fail, but DataLad will adaptive try out other URLs (including changes in hostname, path, or protocol) to enable the ora-remote and retrieve file contents.

Just as expected, the subdatasets are not pre-installed. How will subdataset installation work for datasets that exist in a RIA store as well, like `midterm_project`? Just as with any other subdataset! DataLad cleverly handles subdataset installations from RIA stores in the background: The location of the subdataset in the RIA store is discovered and used automatically:

```
$ datalad get -n midterm_project
[INFO] Cloning dataset to Dataset(/home/me/dl-101/myclone/midterm_project)
[INFO] Attempting to clone from file:///home/me/myriastore/fea/0e9c7-7932-41d5-8d57-
↪97ac9ac6755a to /home/me/dl-101/myclone/midterm_project
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/myclone/midterm_project)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Configure additional publication dependency on "ria-backup-storage"
install(ok): /home/me/dl-101/myclone/midterm_project (dataset) [Installed subdataset in␣
↪order to get /home/me/dl-101/myclone/midterm_project]
```

More technical insights into the automatic `ria+` URL generation are outlined in the findoutmore below:

> 💡 **M15.14 On cloning datasets with subdatasets from RIA stores**
>
> The usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433) details a RIA-store based publication of a large dataset, split into a nested dataset hierarchy with about 4500 subdatasets in total. But how can links to subdatasets work, if datasets in a RIA store are stored in a flat hierarchy, with no nesting?
> The key to this lies in flexibly regenerating subdataset's URLs based on their ID and a path to the RIA store. The **datalad get** command is capable of generating RIA URLs to subdatasets on its own, if the higher level dataset contains a `datalad get` configuration on `subdataset-source-candidate-origin` that points to the RIA store the subdataset is published in. Here is how the `.datalad/config` configuration looks like for the top-level dataset of the HCP dataset[393]:
>
> ```
> [datalad "get"]
>     subdataset-source-candidate-origin = "ria+http://store.datalad.org#{id}"
> ```

💡 With this configuration, a **datalad get** can use the URL and insert the dataset ID in question into the {id} placeholder to clone directly from the RIA store.

This configuration is automatically added to a dataset that is cloned from a RIA store, but it can also be done by hand with a **git config** command[405].

---

[393] https://github.com/datalad-datasets/human-connectome-project-openaccess

[405] To re-read on configuring datasets with the **git config**, go back to sections *DIY configurations* (page 112) and *More on DIY configurations* (page 117).

Beyond straightforward access to datasets, RIA stores also allow very fine-grained cloning operations: Datasets in RIA stores can be cloned in specific versions.

💡 **M15.15 Cloning specific dataset versions**

Optionally, datasets can be cloned in a specific version, such as a TAG or BRANCH by appending @<version-identifier> after the dataset ID or the dataset alias. Here is how to clone the BIDS[394] version of the structural preprocessed subset of the HCP dataset[395] that exists on the branch bids of this dataset:

```
$ datalad clone ria+http://store.datalad.org#~hcp-structural-preprocessed@bids
```

If you are interested in finding out how this dataset came into existence, checkout the use case *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433).

---

[394] https://bids.neuroimaging.io/

[395] https://github.com/datalad-datasets/hcp-structural-preprocessed

Updating datasets works with the **datalad update** and **datalad update --merge** commands introduced in chapter *Collaboration* (page 92). And because a RIA store hosts BARE GIT REPOSITORIES, collaborating becomes easy. Anyone with access can clone the dataset from the store, add changes, and push them back – this is the same workflow as for datasets hosted on sites such as GITHUB, GITLAB, or GIN.

Permission management

In order to limit access or give access to datasets in datastores, permissions can be set at the time of RIA sibling creation with the --shared option. If it is given, this option configures the permissions in the RIA store for multi-users access. Possible values for this option are identical to those of git init --shared and are described in its documentation[396]. In order for the dataset to be accessible to everyone, for example, --shared all could be specified. If access should be limited to a particular Unix group[397] (--shared group), the group name needs to be specified with the --group option.

Configurations and tricks to hide technical layers

In setups with a central, DataLad-centric data management, in order to spare users knowing about RIA stores, custom configurations can be distributed via DataLad's run-procedures to simplify workflows further and hide the technical layers of the RIA setup. For example, custom

---

[396] https://git-scm.com/docs/git-init#Documentation/git-init.txt---sharedfalsetrueumaskgroupallworldeverybody 0xxx

[397] https://en.wikipedia.org/wiki/File_system_permissions#Traditional_Unix_permissions

procedures provided at dataset creation could automatically perform a sibling setup in a RIA store, and also create an associated GitLab repository with a publication dependency to the RIA store to ease publishing data or cloning the dataset. The usecase *Building a scalable data storage for scientific computing* (page 443) details the setup of RIA stores in a scientific institute and demonstrates this example.

To simplify repository access beyond using aliases, the datasets stored in a RIA store can be installed under human-readable names in a single superdataset. Cloning the superdataset exposes the underlying datasets under a non-dataset-ID name. Users can thus get data from datasets hosted in a datastore without any knowledge about the dataset IDs or the need to construct ria+ URLs, just as it was done in the usecases *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433) and *Building a scalable data storage for scientific computing* (page 443). From a user's perspective, the RIA store would thus stay completely hidden.

Standard maintenance tasks by data stewards with knowledge about RIA stores and access to it can be performed easily or even in an automated fashion. The usecase *Building a scalable data storage for scientific computing* (page 443) showcases some examples of those operations.

Summary

RIA stores are useful, lean, and undemanding storage locations for DataLad datasets. Their properties make them suitable solutions to back-up, central data management, or collaboration use cases. They can be set up with minimal effort, and the few technical details a user may face such as cloning from DATASET IDs can be hidden with minimal configurations of the store like aliases or custom procedures.

15.6 Prioritizing subdataset clone locations

When obtaining a superdataset, the subdatasets it contains can have multiple sources. Depending on the use case and precise context, different sources, sometimes referred to as "clone candidates", are more or less "useful". By attaching *costs* to subdataset clone candidates, one can gain precise control over the locations from which subdatasets are retrieved, and the order in which retrieval is attempted. This can create a more flawless and less error-prone user experience as well as speedier dataset installations.

Clone candidates

Let's first exemplify how a dataset can have several clone candidate locations. Consider the case of the `midterm_project` subdataset that was created during the "Basics" part of the handbook: Initially, as this dataset was created as a subdataset of `DataLad-101`, its submodule entry in `DataLad-101/.gitmodules`[407] was a relative path (`./midterm_project`). After it was published to GITHUB in the section on *YODA-compliant data analysis projects* (page 143), this dataset had a second clone candidate location: A URL to its GitHub repository. A third location, finally, was created when publishing the dataset to the RIA store in the previous section *Remote Indexed Archives for dataset storage and backup* (page 294). This makes three locations from where the `midterm_project` subdataset could potentially be obtained from.

Each of these locations can be encoded in the superdataset's `.gitmodules` file, but `.gitmodules` can encode only a single clone candidate. Many use cases, however, benefit from or even require access to several clone candidates. Consider the problem highlighted in *Subdataset publishing* (page 212):

When the `DataLad-101` dataset was published to GIN in section *Walk-through: Dataset hosting on GIN* (page 208), the `.gitmodules` entry of the `midtermproject` subdataset was still a relative path (`./midterm_project`). While this relative path resolves locally on the same machine `DataLad-101` was created on, it does not resolve on GIN. Cloning `DataLad-101` recursively with `midterm_project` thus works when cloned locally from a path, but not when cloned from Gin.

Back in section *Walk-through: Dataset hosting on GIN* (page 208), this problem was fixed by replacing the relative path in `.gitmodules` with the URL to the dataset sibling on GitHub. But a more convenient solution would be to have several known locations for subdatasets that are attempted in succession – if cloning from a local path fails, try the GitHub URL, and then the RIA store, and so forth. Therefore, other than the `.gitmodules` entry, a dataset can encode other clone candidate sources with a configuration variable as well. Here is an overview on where subdataset clone candidates can be found:

1. Without any additional configuration, a subdataset is either registered underneath its superdataset with a relative path (if it was originally created in this dataset), or from the path or URL it was originally installed from. This is recorded in the `.gitmodules` file of the superdataset.

2. Alternatively, subdataset source candidates can be configured under the configuration variable `datalad.get.subdataset-source-candidate-<name>`, where `<name>` is an arbitrary identifier, within either `.datalad/config` (if the configuration should stick with the dataset) or `.git/config` (if it should only apply to the dataset, but not its SIBLINGs or clones).

A concrete example of a clone candidate configuration as well as further details can be found in the next paragraph.

Clone candidate priority

We have established that subdatasets can come from several sources. Let's now motivate *why* it might be useful to prioritize one subdataset clone location over another one.

---

[407] To re-read about `.gitmodules` files and their contents, please go back to section *More on DIY configurations* (page 117).

Consider a hierarchy of datasets that exist in several locations, for example one REMOTE IN-DEXED ARCHIVE (RIA) STORE *with* a storage special remote[408], and one without a special remote. The topmost superdataset is published to a human-readable and accessible location such as GITHUB or GITLAB, and should be configured to always clone subdatasets from the RIA store *with* the storage special remote, even if it was originally created with subdatasets from the RIA store with no storage sibling. In order to be able to retrieve subdataset *data* from the subdatasets after cloning the hierarchy of datasets, the RIA store with the storage special remote needs to be configured as a clone candidate. Importantly, it should not only be configured as one alternative, but it should be configured as the first location to try to clone from – else, cloning from the wrong RIA store could succeed and prevent any configured second clone candidate location from being tried.

> ⚠️ **Use case for clone priorities**
>
> The most likely use case for such a scenario is in the case of centrally managed data with data administrators that provide and manage the data for their users.

The priority of subdataset clone locations is configured by attaching a *cost* to a source candidate `<name>`. The cost is a three digit value (range `000-999`), and the lower cost of a candidate, the higher its priority, i.e., the candidate with the lowest cost is attempted first. In order to prefer any particular RIA store for subdataset cloning, one could configure the superdataset with the following command[409]:

```
$ git config -f .datalad/config datalad.get.subdataset-source-candidate-
→000mypreferredRIAstore ria+http://store.datalad.org#{id}
```

where `mypreferredRIAstore` is the (arbitrary) `<name>` of the source candidate, and the `000` prefix is the (lowest possible) cost. Such a configuration will ensure that the first location any subdataset is attempted to be installed from is the RIA store at `store.datalad.org`. Only if the dataset is not found in there under its ID, other sources are tried. Note that in the case where no cost is provided together with the candidate name, a default cost of `700` is used.

> 💡 **M15.16 What are the "default" costs for preexisting clone candidates?**
>
> The following list provides and overview of which locations are attempted for cloning and their associated costs:
> - `500` for the superdatasets' remote URL + submodule path
> - `600` for the configured submodule URL in `.gitmodules`
> - `700` for any unprioritized `datalad.get.subdataset-source-candidate` config
> - `900` for the local subdataset path

With the `datalad.get.subdataset-source-candidate` configuration any number of (differently named) clone candidates can be set and prioritized. This allows precise access control over subdataset clone locations, and can – depending on how many subdataset locations are known and functional – speed up dataset installation.

---

[408] To re-read about RIA stores and their ORA special remote storage siblings, please take a look at the section *Remote Indexed Archives for dataset storage and backup* (page 294).

[409] If you are unsure how the **git config** command works, please check out section *DIY configurations* (page 112).

Placeholders

Instead of adding configurations with precise URLs you can also make use of templates with placeholders to configure clone locations more flexibly. A placeholder takes the form {placeholdername} and can reference any property that can be inferred from the parent dataset's knowledge about the target superset, specifically any subdataset information that exists as a key-value pair within .gitmodules. For convenience, an existing *datalad-id* record is made available under the shortened name *id*. In all likelihood, the list of available placeholders will be expanded in the future. Do you have a usecase and need a specific placeholder? Reach out to us[406], we may be able to add the placeholders you need!

When could this be useful? For an example, consider how the clone candidate configuration above did not specify a concrete dataset in the RIA store, but used the {id} placeholder, which will expand to the subdataset's DATASET ID upon cloning. This ensures that the clone locations point to the same RIA store, but stay flexible and dataset-specific. You could configure a specific path or URL as a clone location, but this configuration is applied to *all* subdatasets. Thus, whenever more than one subdataset exists in a superdataset, make sure to not provide a clone candidate configuration to a single, particular subdataset, as this could jeopardize the clone location of any other subdataset.

15.7 Subsample datasets using datalad copy-file

If there is a need for a dataset that contains only a subset of files of one or more other dataset, it can be helpful to create subsamples special-purpose datasets with the **datalad copy-file** command (datalad-copy-file manual). This command is capable of transferring files from different datasets or locations outside of a dataset into a new dataset, unlocking them if necessary, and preserving and copying their availability information. As such, the command is a superior, albeit more technical alternative to *copying dereferenced files out of datasets* (page 486).

> ⚠️ **Version requirement for datalad copy-file**
>
> **datalad copy-file** requires DataLad version 0.13.0 or higher.

This section demonstrates the command based on a published data, a subset of the Human Connectome Project dataset that is subsampled for structural connectivity analysis. This dataset can be found on GitHub at github.com/datalad-datasets/hcp-structural-connectivity[410].

Copy-file in action with the HCP dataset

Consider a real-life example: A large number of scientists use the human connectome project (HCP) dataset[411] for structural connectivity analyses[412]. This dataset contains data from more than 1000 subjects, and exceeds 80 million files. As such, as explained in more detail in the chapter *Go big or go home* (page 323), it is split up into a hierarchy of roughly 4500 sub-datasets[415]. The installation of all subdatasets takes around 90 minutes, if parallelized, and a

---

[406] https://github.com/datalad/datalad/issues/new

[410] https://github.com/datalad-datasets/hcp-structural-connectivity

[411] https://github.com/datalad-datasets/human-connectome-project-openaccess

[412] https://en.wikipedia.org/wiki/Brain_connectivity_estimators

[415] You can read about the human connectome dataset in the usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433).

complete night if performed serially. However, for a structural connectivity analysis, only eleven files per subject are relevant:

- `<sub>/T1w/Diffusion/nodif_brain_mask.nii.gz`
- `<sub>/T1w/Diffusion/bvecs`
- `<sub>/T1w/Diffusion/bvals`
- `<sub>/T1w/Diffusion/data.nii.gz`
- `<sub>/T1w/Diffusion/grad_dev.nii.gz`
- `<sub>/unprocessed/3T/T1w_MPR1/*_3T_BIAS_32CH.nii.gz`
- `<sub>/unprocessed/3T/T1w_MPR1/*_3T_AFI.nii.gz`
- `<sub>/unprocessed/3T/T1w_MPR1/*_3T_BIAS_BC.nii.gz`
- `<sub>/unprocessed/3T/T1w_MPR1/*_3T_FieldMap_Magnitude.nii.gz`
- `<sub>/unprocessed/3T/T1w_MPR1/*_3T_FieldMap_Phase.nii.gz`
- `<sub>/unprocessed/3T/T1w_MPR1/*_3T_T1w_MPR1.nii.gz`

In order to spare others the time and effort to install thousands of subdatasets, a one-time effort can create and publish a subsampled, single dataset of those files using the **datalad copy-file** command.

datalad copy-file is able to copy files with their availability metadata into other datasets. The content of the files does not need to be retrieved in order to do this. Because the subset of relevant files is small, all structural connectivity related files can be copied into a single dataset. This speeds up the installation time significantly, and reduces the confusion that the concept of subdatasets can bring to DataLad novices. The result is a dataset with a subset of files (following the original directory structure of the HCP dataset), created reproducibly with complete provenance capture. Access to the files inside of the subsampled dataset works via valid AWS credentials just as it does for the full dataset[415].

The Basics of copy-file

This short demonstration gives an overview of the functionality of **datalad copy-file** - Feel free to follow along by copy-pasting the commands into your terminal. Let's start by cloning a dataset to work with:

```
$ datalad clone https://github.com/datalad-datasets/human-connectome-project-openaccess.
↪git hcp
[INFO] Cloning dataset to Dataset(/home/me/dl-101/HPC/hcp)
[INFO] Attempting to clone from https://github.com/datalad-datasets/human-connectome-
↪project-openaccess.git to /home/me/dl-101/HPC/hcp
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/dl-101/HPC/hcp)
install(ok): /home/me/dl-101/HPC/hcp (dataset)
```

In order to use **copy-file**, we need to install a few subdatasets, and thus install 9 subject subdatasets recursively. Note that we don't retrieve any data, using `-n/--no-data`. (The output of this command is omitted – it is quite lengthy as 36 subdatasets are being installed)

```
$ cd hcp
$ datalad get -n -r HCP1200/130*
[INFO] Cloning dataset to Dataset(/home/me/dl-101/HPC/hcp/HCP1200/130013)
```

Afterwards, we can create a new dataset to copy any files into. This dataset will later hold the relevant subset of the data in the HCP dataset.

```
$ cd ..
$ datalad create dataset-to-copy-to
[INFO] Creating a new annex repo at /home/me/dl-101/HPC/dataset-to-copy-to
[INFO] scanning for unlocked files (this may take some time)
create(ok): /home/me/dl-101/HPC/dataset-to-copy-to (dataset)
```

With the prerequisites set up, we can start to copy files. The command **datalad copy-file** works as follows: By providing a path to a file to be copied (which can be annex'ed, not annex'ed, or not version-controlled at all) and either a second path (the destination path), a target directory inside of a dataset, or a dataset specification, **datalad copy-file** copies the file and all of its availability metadata into the specified dataset. Let's copy a single file (hcp/HCP1200/130013/T1w/Diffusion/bvals) from the hcp dataset into dataset-to-copy-to:

```
$ datalad copy-file \
   hcp/HCP1200/130013/T1w/Diffusion/bvals  \
   -d dataset-to-copy-to
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130013/T1w/Diffusion/bvals [/home/me/dl-
↪101/HPC/dataset-to-copy-to/bvals]
save(ok): . (dataset)
action summary:
  copy_file (ok: 1)
  save (ok: 1)
```

When the -d/--dataset argument is provided instead of a target directory or a destination path, the copied file will be *saved* in the new dataset. If a target directory or a destination path is given for a file, however, the copied file will be not be saved:

```
$ datalad copy-file \
   hcp/HCP1200/130013/T1w/Diffusion/bvecs \
   -t dataset-to-copy-to
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130013/T1w/Diffusion/bvecs [/home/me/dl-
↪101/HPC/dataset-to-copy-to/bvecs]
```

Note that instead of a as dataset, we specify it as a target path, and how the file is added, but not saved afterwards:

```
$ cd dataset-to-copy-to
$ datalad status
    added: bvecs (file)
```

Providing a second path as a *destination* path allows one to copy the file under a different name, but it will also not save the new file in the destination dataset unless -d/--dataset is specified as well:

```
$ datalad copy-file \
   hcp/HCP1200/130013/T1w/Diffusion/bvecs \
   dataset-to-copy-to/anothercopyofbvecs
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130013/T1w/Diffusion/bvecs [/home/me/dl-
↪101/HPC/dataset-to-copy-to/anothercopyofbvecs]
```

```
$ cd dataset-to-copy-to
$ datalad status
```

```
    added: anothercopyofbvecs (file)
    added: bvecs (file)
```

Those were the minimal basics of the command syntax - the original location, a specification where the file should be copied to, and an indication if the file should be saved or not. Let's save those two unsaved files:

```
$ datalad save
save(ok): . (dataset)
```

With the `-r`/`--recursive` flag enabled, the command can copy complete *subdirectory* (not subdataset!) hierarchies – Let's copy a complete directory, and save it in its target dataset:

```
$ cd ..
$ datalad copy-file hcp/HCP1200/130114/T1w/Diffusion/* \
 -r \
 -d dataset-to-copy-to \
 -t dataset-to-copy-to/130114/T1w/Diffusion
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/bvals [/home/me/dl-
↪101/HPC/dataset-to-copy-to/130114/T1w/Diffusion/bvals]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/bvecs [/home/me/dl-
↪101/HPC/dataset-to-copy-to/130114/T1w/Diffusion/bvecs]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/data.nii.gz [/home/me/
↪dl-101/HPC/dataset-to-copy-to/130114/T1w/Diffusion/data.nii.gz]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_parameters [/home/me/dl-101/HPC/dataset-to-copy-to/130114/T1w/
↪Diffusion/eddylogs/eddy_unwarped_images.eddy_parameters]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_outlier_n_stdev_map [/home/me/dl-101/HPC/dataset-to-copy-to/130114/
↪T1w/Diffusion/eddylogs/eddy_unwarped_images.eddy_outlier_n_stdev_map]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_outlier_map [/home/me/dl-101/HPC/dataset-to-copy-to/130114/T1w/
↪Diffusion/eddylogs/eddy_unwarped_images.eddy_outlier_map]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_outlier_n_sqr_stdev_map [/home/me/dl-101/HPC/dataset-to-copy-to/
↪130114/T1w/Diffusion/eddylogs/eddy_unwarped_images.eddy_outlier_n_sqr_stdev_map]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_post_eddy_shell_alignment_parameters [/home/me/dl-101/HPC/dataset-
↪to-copy-to/130114/T1w/Diffusion/eddylogs/eddy_unwarped_images.eddy_post_eddy_shell_
↪alignment_parameters]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_outlier_report [/home/me/dl-101/HPC/dataset-to-copy-to/130114/T1w/
↪Diffusion/eddylogs/eddy_unwarped_images.eddy_outlier_report]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_movement_rms [/home/me/dl-101/HPC/dataset-to-copy-to/130114/T1w/
↪Diffusion/eddylogs/eddy_unwarped_images.eddy_movement_rms]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/eddylogs/eddy_
↪unwarped_images.eddy_restricted_movement_rms [/home/me/dl-101/HPC/dataset-to-copy-to/
↪130114/T1w/Diffusion/eddylogs/eddy_unwarped_images.eddy_restricted_movement_rms]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/grad_dev.nii.gz [/
↪home/me/dl-101/HPC/dataset-to-copy-to/130114/T1w/Diffusion/grad_dev.nii.gz]
copy_file(ok): /home/me/dl-101/HPC/hcp/HCP1200/130114/T1w/Diffusion/nodif_brain_mask.nii.
↪gz [/home/me/dl-101/HPC/dataset-to-copy-to/130114/T1w/Diffusion/nodif_brain_mask.nii.gz]
save(ok): . (dataset)
action summary:
```

Chapter 15. Advanced options

```
copy_file (ok: 13)
save (ok: 1)
```

Here is how the dataset that we copied files into looks like at the moment:

```
$ tree dataset-to-copy-to
dataset-to-copy-to
├── 130114
│   └── T1w
│       └── Diffusion
│           ├── bvals -> ../../../.git/annex/objects/w8/VX/MD5E-s1344--
→4c9ca43cc986f388bcf716b4ba7321cc/MD5E-s1344--4c9ca43cc986f388bcf716b4ba7321cc
│           ├── bvecs -> ../../../.git/annex/objects/61/80/MD5E-s9507--
→24793fb936e9e18419325af9b6152458/MD5E-s9507--24793fb936e9e18419325af9b6152458
│           ├── data.nii.gz -> ../../../.git/annex/objects/K0/mJ/MD5E-s1468805393--
→f8077751ddc2802a853d1199ff762a00.nii.gz/MD5E-s1468805393--
→f8077751ddc2802a853d1199ff762a00.nii.gz
│           ├── eddylogs
│           │   ├── eddy_unwarped_images.eddy_movement_rms -> ../../../../.git/annex/
→objects/xX/GF/MD5E-s15991--287c3e06ece5b883a862f79c478b7b69/MD5E-s15991--
→287c3e06ece5b883a862f79c478b7b69
│           │   ├── eddy_unwarped_images.eddy_outlier_map -> ../../../../.git/annex/
→objects/87/Xx/MD5E-s127363--919aed21eb51a77ca499cdc0a5560592/MD5E-s127363--
→919aed21eb51a77ca499cdc0a5560592
│           │   ├── eddy_unwarped_images.eddy_outlier_n_sqr_stdev_map -> ../../../../.git/
→annex/objects/PP/GX/MD5E-s523738--1bd90e1e7a86b35695d8039599835435/MD5E-s523738--
→1bd90e1e7a86b35695d8039599835435
│           │   ├── eddy_unwarped_images.eddy_outlier_n_stdev_map -> ../../../../.git/
→annex/objects/qv/0F/MD5E-s520714--f995a46ec8ddaa5c7b33d71635844609/MD5E-s520714--
→f995a46ec8ddaa5c7b33d71635844609
│           │   ├── eddy_unwarped_images.eddy_outlier_report -> ../../../../.git/annex/
→objects/Xq/xV/MD5E-s10177--2934d2c7b316b86cde6d6d938bb3da37/MD5E-s10177--
→2934d2c7b316b86cde6d6d938bb3da37
│           │   ├── eddy_unwarped_images.eddy_parameters -> ../../../../.git/annex/
→objects/60/gf/MD5E-s141201--9a94e9fa805446ddb5ff8f76207fc1d2/MD5E-s141201--
→9a94e9fa805446ddb5ff8f76207fc1d2
│           │   ├── eddy_unwarped_images.eddy_post_eddy_shell_alignment_parameters -> ../.
→./../../.git/annex/objects/kJ/0W/MD5E-s2171--c2e0deca2a5e84d119002032d87cd762/MD5E-
→s2171--c2e0deca2a5e84d119002032d87cd762
│           │   └── eddy_unwarped_images.eddy_restricted_movement_rms -> ../../../../.git/
→annex/objects/6K/X6/MD5E-s16134--5321d11df307f8452c8a5e92647ec73a/MD5E-s16134--
→5321d11df307f8452c8a5e92647ec73a
│           ├── grad_dev.nii.gz -> ../../../.git/annex/objects/zz/51/MD5E-s46820650--
→13be960cd99e48e21e25635d1390c1c5.nii.gz/MD5E-s46820650--
→13be960cd99e48e21e25635d1390c1c5.nii.gz
│           └── nodif_brain_mask.nii.gz -> ../../../.git/annex/objects/0Q/Kk/MD5E-s67280--
→9042713a11d557df58307ba85d51285a.nii.gz/MD5E-s67280--9042713a11d557df58307ba85d51285a.
→nii.gz
├── anothercopyofbvecs -> .git/annex/objects/X0/Vg/MD5E-s9507--
→f4cf263de8c3fb11f739467bf15e80ec/MD5E-s9507--f4cf263de8c3fb11f739467bf15e80ec
├── bvals -> .git/annex/objects/Fj/Wg/MD5E-s1344--843688799692be0ab485fe746e0f9241/MD5E-
→s1344--843688799692be0ab485fe746e0f9241
└── bvecs -> .git/annex/objects/X0/Vg/MD5E-s9507--f4cf263de8c3fb11f739467bf15e80ec/MD5E-
→s9507--f4cf263de8c3fb11f739467bf15e80ec

4 directories, 16 files
```

Importantly, all of the copied files had yet unretrieved contents. The copy-file process, however, also copied the files' availability metadata to their new location. Retrieving file contents works just as it would in the full HCP dataset via **datalad get** (the authentication step is omitted in the output below):

```
$ cd dataset-to-copy-to
$ datalad get bvals anothercopyofbvecs 130114/T1w/Diffusion/eddylogs/eddy_unwarped_images.
↪eddy_parameters
get(ok): bvals (file) [from datalad...]
get(ok): anothercopyofbvecs (file) [from datalad...]
get(ok): 130114/T1w/Diffusion/eddylogs/eddy_unwarped_images.eddy_parameters (file) [from↲
↪datalad...]
action summary:
  get (ok: 3)
```

What's especially helpful for automation of this operation is that **copy-file** can take source and (optionally) destination paths from a file or from STDIN with the option --specs-from <source>. In the case of specifications from a file, <source> is a path to this file.

In order to use stdin for specification, such as the output of a find command that is piped into **datalad copy-file** with a Unix pipe (|)[413], <source> needs to be a dash (-). Below is an example find command:

```
$ cd hcp
$ find HCP1200/130013/T1w/ -maxdepth 1 -name T1w*.nii.gz
HCP1200/130013/T1w/T1w_acpc_dc.nii.gz
HCP1200/130013/T1w/T1w_acpc_dc_restore_1.25.nii.gz
HCP1200/130013/T1w/T1wDividedByT2w.nii.gz
HCP1200/130013/T1w/T1wDividedByT2w_ribbon.nii.gz
HCP1200/130013/T1w/T1w_acpc_dc_restore_brain.nii.gz
HCP1200/130013/T1w/T1w_acpc_dc_restore.nii.gz
```

This uses find to get a list of all files matching the specified pattern in the specified directory. And here is how the outputted paths can be given as source paths to **datalad copy-file**, copying all of the found files into a new dataset:

```
# inside of hcp
$ find HCP1200/130013/T1w/ -maxdepth 1 -name T1w*.nii.gz \
  | datalad copy-file -d ../dataset-to-copy-to --specs-from -
copy_file(ok): HCP1200/130013/T1w/T1w_acpc_dc.nii.gz [/home/me/dl-101/HPC/dataset-to-copy-
↪to/T1w_acpc_dc.nii.gz]
copy_file(ok): HCP1200/130013/T1w/T1w_acpc_dc_restore_1.25.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/T1w_acpc_dc_restore_1.25.nii.gz]
copy_file(ok): HCP1200/130013/T1w/T1wDividedByT2w.nii.gz [/home/me/dl-101/HPC/dataset-to-
↪copy-to/T1wDividedByT2w.nii.gz]
copy_file(ok): HCP1200/130013/T1w/T1wDividedByT2w_ribbon.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/T1wDividedByT2w_ribbon.nii.gz]
copy_file(ok): HCP1200/130013/T1w/T1w_acpc_dc_restore_brain.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/T1w_acpc_dc_restore_brain.nii.gz]
copy_file(ok): HCP1200/130013/T1w/T1w_acpc_dc_restore.nii.gz [/home/me/dl-101/HPC/dataset-
↪to-copy-to/T1w_acpc_dc_restore.nii.gz]
save(ok): . (dataset)
action summary:
  copy_file (ok: 6)
  save (ok: 1)
```

---

[413] https://en.wikipedia.org/wiki/Pipeline_(Unix)

To preserve the directory structure, a target directory (`-t ../dataset-to-copy-to/130013/T1w/`) or a destination path could be given, because the above command copied all files into the root of `dataset-to-copy-to`:

```
$ ls ../dataset-to-copy-to
130114
anothercopyofbvecs
bvals
bvecs
T1w_acpc_dc.nii.gz
T1w_acpc_dc_restore_1.25.nii.gz
T1w_acpc_dc_restore_brain.nii.gz
T1w_acpc_dc_restore.nii.gz
T1wDividedByT2w.nii.gz
T1wDividedByT2w_ribbon.nii.gz
```

With this trick, you can use simple search commands to assemble a list of files as a `<source>` for **copy-file**: simply create a file or a command like `find` that specifies tho relevant files or directories line-wise. `--specs-from` can take information on both `<source>` and `<destination>`, though.

Specify files with source AND destination paths for –specs-from

Specifying source *and* destination paths comes with a twist: Source and destination paths need to go into the same line, but need to be separated by a nullbyte[414]. This is not a straightforward concept, but trying it out and seeing it in action will help.

One way it can be done is by using the stream editor SED. Here is how to pipe source AND destination paths into **datalad copy-file**:

```
$ find HCP1200/130518/T1w/ -maxdepth 1 -name T1w*.nii.gz \
  | sed -e 's#\(HCP1200\)\(.*\)#\1\2\x0../dataset-to-copy-to\2#' \
  | datalad copy-file -d ../dataset-to-clone-to -r --specs-from -
```

As always, the regular expressions used for sed are a bit hard to grasp upon first sight. Here is what this command does:

- In general, SED's **s** (substitute) command will take a string specified between the first set of #'s (`\(HCP1200\)\(.*\)`) and replace it with what is between the second and third # (`\1\2\x0\2`).

- The first part splits the paths `find` returns (such as `HCP1200/130518/T1w/T1w_acpc_dc.nii.gz`) into two groups:

    - The start of the path (`HCP1200`), and

    - the remaining path (`/130518/T1w/T1w_acpc_dc.nii.gz`).

    - The second part then prints the first and the second group (`\1\2`, the source path), a nullbyte (`\x0`), and a relative path to the destination dataset together with the second group only (`../dataset-to-copy-to\2`, the destination path).

Here is how an output of `find` piped into `sed` looks like:

```
$ find HCP1200/130518/T1w -maxdepth 1 -name T1w*.nii.gz \
```

---

[414] https://en.wikipedia.org/wiki/Null_character

```
      | sed -e 's#\(HCP1200\)\(.*\)#\1\2\x0../dataset-to-copy-to\2#'
HCP1200/130518/T1w/T1w_acpc_dc.nii.gz../dataset-to-copy-to/130518/T1w/T1w_acpc_dc.
↪nii.gz
HCP1200/130518/T1w/T1w_acpc_dc_restore_1.25.nii.gz../dataset-to-copy-to/130518/
↪T1w/T1w_acpc_dc_restore_1.25.nii.gz
HCP1200/130518/T1w/T1wDividedByT2w.nii.gz../dataset-to-copy-to/130518/T1w/
↪T1wDividedByT2w.nii.gz
HCP1200/130518/T1w/T1wDividedByT2w_ribbon.nii.gz../dataset-to-copy-to/130518/T1w/
↪T1wDividedByT2w_ribbon.nii.gz
HCP1200/130518/T1w/T1w_acpc_dc_restore_brain.nii.gz../dataset-to-copy-to/130518/
↪T1w/T1w_acpc_dc_restore_brain.nii.gz
HCP1200/130518/T1w/T1w_acpc_dc_restore.nii.gz../dataset-to-copy-to/130518/T1w/T1w_
↪acpc_dc_restore.nii.gz
HCP1200/130518/T1w/T1w_acpc_dc_restore_1.05.nii.gz../dataset-to-copy-to/130518/
↪T1w/T1w_acpc_dc_restore_1.05.nii.gz
```

Note how the nullbyte is not visible to the naked eye in the output. To visualize it, you could redirect this output into a file and open it with an editor like VIM. Let's now see a **copy-file** from STDIN in action:

```
$ find HCP1200/130518/T1w -maxdepth 1 -name T1w*.nii.gz \
  | sed -e 's#\(HCP1200\)\(.*\)#\1\2\x0../dataset-to-copy-to\2#' \
  | datalad copy-file -d ../dataset-to-copy-to -r --specs-from -
copy_file(ok): HCP1200/130518/T1w/T1w_acpc_dc.nii.gz [/home/me/dl-101/HPC/dataset-to-copy-
↪to/130518/T1w/T1w_acpc_dc.nii.gz]
copy_file(ok): HCP1200/130518/T1w/T1w_acpc_dc_restore_1.25.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/130518/T1w/T1w_acpc_dc_restore_1.25.nii.gz]
copy_file(ok): HCP1200/130518/T1w/T1wDividedByT2w.nii.gz [/home/me/dl-101/HPC/dataset-to-
↪copy-to/130518/T1w/T1wDividedByT2w.nii.gz]
copy_file(ok): HCP1200/130518/T1w/T1wDividedByT2w_ribbon.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/130518/T1w/T1wDividedByT2w_ribbon.nii.gz]
copy_file(ok): HCP1200/130518/T1w/T1w_acpc_dc_restore_brain.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/130518/T1w/T1w_acpc_dc_restore_brain.nii.gz]
copy_file(ok): HCP1200/130518/T1w/T1w_acpc_dc_restore.nii.gz [/home/me/dl-101/HPC/dataset-
↪to-copy-to/130518/T1w/T1w_acpc_dc_restore.nii.gz]
copy_file(ok): HCP1200/130518/T1w/T1w_acpc_dc_restore_1.05.nii.gz [/home/me/dl-101/HPC/
↪dataset-to-copy-to/130518/T1w/T1w_acpc_dc_restore_1.05.nii.gz]
save(ok): . (dataset)
action summary:
  copy_file (ok: 7)
  save (ok: 1)
```

Done! A complex looking command with regular expressions and unix pipes, but it does powerful things in only a single line.

Copying reproducibly

To capture the provenance of subsampled dataset creation, the **copy-file** command can be wrapped into a **datalad run** call. Here is a sketch how it was done in the structural connectivity subdataset:

Step 1: Create a dataset

```
$ datalad create hcp-structural-connectivity
```

Step 2: Install the full dataset as a subdataset

```
$ datalad clone -d . \
  https://github.com/datalad-datasets/human-connectome-project-openaccess.git \
  .hcp
```

Step 3: Install all subdataset of the full dataset with `datalad get -n -r`

Step 4: Inside of the new dataset, draft a `find` command that returns all 11 desired files, and a subsequent sed substitution command that returns a nullbyte separated source and destination path. For this subsampled dataset, this one would work:

```
$ find .hcp/HCP1200  -maxdepth 5 -path '*/unprocessed/3T/T1w_MPR1/*' -name '*' \
 -o -path '*/T1w/Diffusion/*' -name 'b*' \
 -o -path '*/T1w/Diffusion/*' -name '*.nii.gz' \
 | sed -e 's#\(\.hcp/HCP1200\)\(.*\)#\1\2\x00.\2#' \
```

Step 5: Pipe the results into **`datalad copy-file`**, and wrap everything into a **`datalad run`**. Note that `-d/--dataset` is not specified for **`copy-file`** – this way, **`datalad run`** will save everything in one go at the end.

```
$ datalad run \
  -m "Assemble HCP dataset subset for structural connectivity data. \

    Specifically, these are the files:

 - T1w/Diffusion/nodif_brain_mask.nii.gz
    - T1w/Diffusion/bvecs
    - T1w/Diffusion/bvals
    - T1w/Diffusion/data.nii.gz
    - T1w/Diffusion/grad_dev.nii.gz
    - unprocessed/3T/T1w_MPR1/*_3T_BIAS_32CH.nii.gz
    - unprocessed/3T/T1w_MPR1/*_3T_AFI.nii.gz
    - unprocessed/3T/T1w_MPR1/*_3T_BIAS_BC.nii.gz
    - unprocessed/3T/T1w_MPR1/*_3T_FieldMap_Magnitude.nii.gz
    - unprocessed/3T/T1w_MPR1/*_3T_FieldMap_Phase.nii.gz
    - unprocessed/3T/T1w_MPR1/*_3T_T1w_MPR1.nii.gz

    for each participant. The structure of the directory tree and file names
    are kept identical to the full HCP dataset." \
    "find .hcp/HCP1200  -maxdepth 5 -path '*/unprocessed/3T/T1w_MPR1/*' -name '*' \
      -o -path '*/T1w/Diffusion/*' -name 'b*' \
      -o -path '*/T1w/Diffusion/*' -name '*.nii.gz' \
    | sed -e 's#\(\.hcp/HCP1200\)\(.*\)#\1\2\x00.\2#' \
    | datalad copy-file -r --specs-from -"
```

Step 6: Publish the dataset to GITHUB or similar hosting services to allow others to clone it easily and get fast access to a relevant subset of files.

Afterwards, the slimmed down structural connectivity dataset can be installed completely within seconds. Because of the reduced amount of files it contains, it is easier to transform the data into BIDS format. Such a conversion can be done on a different BRANCH of the dataset. If you have published your subsampled dataset into a RIA store, as it was done with this specific subset, a single command can clone a BIDS-ified, slimmed down HCP dataset for structural connectivity

---

analyses because RIA stores allow cloning of datasets in specific versions (such as a branch or tag as an identifier):

```
$ datalad clone ria+http://store.datalad.org#~hcp-structural-connectivity@bids
```

Summary

`datalad copy-file` is a useful command to create datasets from content of other datasets. Although it requires some Unix-y command line magic, it can be automated for larger tasks, and, when combined with a `datalad run`, produce suitable provenance records of where files have been copied from.

GO BIG OR GO HOME
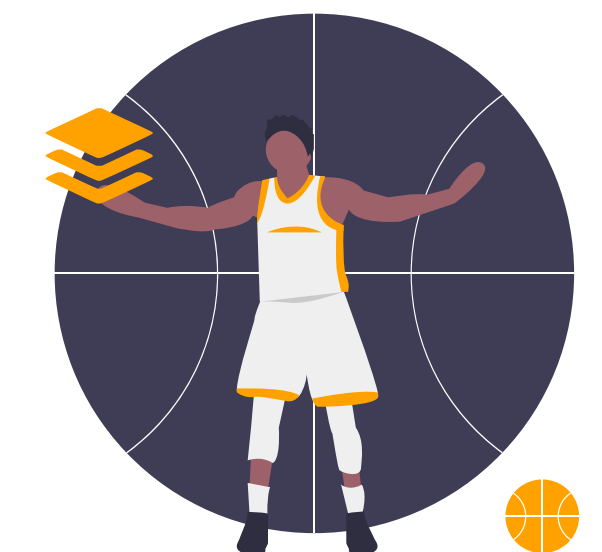
16.1 Going big with DataLad

All chapters throughout the Basics demonstrated "household quantity" examples. Version controlling or analyzing data in datasets with a total size of up to a few hundred GB, with some tens of thousands of files at maximum? Usually, this should work fine. If you want to go beyond this scale, however, you should read this section to learn how to properly scale up. As a general rule, consider this section relevant once you have a usecase in which you would go substantially beyond 100k files in a single dataset.

The contents of this chapter exist thanks to some pioneers that took a leap and deep-dived into gigantic data management challenges. You can read up on some of them in the usecases *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433) and *Building a scalable data storage for scientific computing* (page 443). Based on what we have learned so far from these endeavors, this chapter encompasses principles, advice, and points of reference.

The introduction in this section illustrates the basic caveats when scaling up, and points to benchmarks, rules of thumb, and general solutions. Upcoming sections demonstrate how one can attempt large-scale analyses with DataLad, and how to fix things up when dataset sizes got out of hand. The upcoming chapter *Computing on clusters* (page 331), finally, extends this chapter with advice and examples from large scale analyses on computational clusters.

Why scaling up Git repos can become difficult

You already know that GIT does not scale well with *large* files. As a Git repository stores every version of every file that is added to it, large files that undergo regular modifications can inflate the size of a·project significantly. Depending on how many large files are added to a pure Git repository, this can not only have a devastating impact on the time it takes to clone, fetch, or pull (from) a repository, but also on regular within-repository operations, such as checking the state of the repository or switching branches. Using GIT-ANNEX (either directly, or by using DataLad) can eliminate this issue, but there is a second factor that can prevent scaling up with Git: The *number* of files. One reason for this is that Git performs a large amount of stat system calls[416] (used in `git add` and `git commit`). Repositories can thus suffer greatly if they are swamped with files[424].

Given that DataLad builds up on Git, having datasets with large amounts of files can lead to painfully slow operations[417]. As a general rule of thumb, we will consider single datasets with 100k files or more as "big" for the rest of this chapter. Starting at about this size we can begin to see performance issues in datasets. Bench marking in DataLad datasets with varying, but large amounts of tiny files on different file systems and different git-annex repository versions show that a mere `datalad save` or `datalad status` command can take from 15 minutes up to several hours. Its neither fun nor feasible to work with performance drops like this – so how can this be avoided?

General advice: Use several subdatasets

The general set-up for publishing or version controlling data in a scalable way is to make use of subdatasets. Instead of a single dataset with 1 million files, have 20, for example, with 50.000 files each, and link them as subdataset. This will split the amount of files that need to be handled across several datasets, and, at the same time, it also alleviates strain on the file system that would arise if large amounts of files are kept in single directories.

How would that look like for a large scale dataset? In the use case *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433), 80 million files with neuroscientific data from about 1200 participants are split into roughly 4500 subdatasets based on directory structure. Each participant directory is a subdataset, and it contains several more subdatasets, depending on how much data modalities are available. A similar approach was chosen for the Datalad UKbiobank extension[418] that can enable to obtain and version control imaging releases of the up to 100000 participants of the UKbiobank project[419].

"But why use DataLad for this?" In principle, using many instead of a single repository/dataset for large amounts of files is a measure that can be implemented with any of the tools involved, be it Git, git-annex, or DataLad. What makes using DataLad well-suited for such a scaling approach and distinguishes it from Git and git-annex, is that it is way easier to link datasets and

---

[416] https://en.wikipedia.org/wiki/Stat_(system_call)

[424] For example: A Git repository with more than a million (albeit tiny) files takes hours and hours to merely create[425], if standard Git workflows are used. This post[426] contains an entertaining description of what happens if one attempts to create a Git repository with 6.5 million files – up to the point when some Git commands stop working.

[425] https://www.monperrus.net/martin/one-million-files-on-git-and-github

[426] https://breckyunits.com/building-a-treebase-with-6-point-5-million-files.html

[417] https://github.com/datalad/datalad/issues/3869

[418] https://github.com/datalad/datalad-ukbiobank

[419] https://www.ukbiobank.ac.uk/

to operate across subdataset boundaries recursively with the nesting capabilities[427] of DataLad. Git provides functionality for nested repositories (so-called submodules, also used by DataLad underneath the hood), but the workflows are by far not as smooth. For a direct comparison between working with nested datasets and nested Git repositories, take a look at this demo[420].

How far does this scale? In preparation for assembling a complete UKBiobank dataset, simulations of datasets with 40k and 100k subdatasets ran successfully.

> **M16.1 How do simulations like this work?**
>
> With shell scripts such as this:
>
> ```bash
> #!/bin/bash
> set -x
>
> # build a dummy subdataset to be referenced 40k times:
> datalad create dummy_sub
> echo "whatever" > dummy_sub/some_file
> datalad save -d dummy_sub
>
> sub_id=$(datalad -f "{infos[dataset][id]}"  wtf -d dummy_sub)
> sub_commit=$(git -C dummy_sub show --no-patch --format=%H)
>
>
> # the actual super dataset and use some config procedure to get
> # an initial history
> datalad create -c yoda dummy_super_40k
>
> cd dummy_super_40k
>
> for ((i=1;i<=100000;i++)); do
>     git config -f .gitmodules "submodule.sub$i.path" "sub$i";
>     git config -f .gitmodules "submodule.sub$i.url" ../dummy_sub;
>     git config -f .gitmodules "submodule.sub$i.datalad-id" "$sub_id";
>     git update-index --add --replace --cacheinfo 160000 "$sub_commit" "sub$i";
> done;
>
> git add .gitmodules
> git commit -m "Add submodules"
> ```
>
> Note that this way of simulating subdatasets is speedier and simplified, because instead of cloning subdatasets, it makes use of Git's update-index[421] command and records the subdatasets by committing manual changes to .gitmodules.
>
> ---
> [421] https://git-scm.com/docs/git-update-index

Do note, however, that these numbers of subdatasets may well exhaust your file system's subdirectory limit (commonly at 64k).

---
[427] To reread on nesting DataLad datasets, check out sections *Dataset nesting* (page 53) and *More on Dataset nesting* (page 164)

[420] https://youtu.be/Yrg6DgOcbPE?t=350

Tool-specific and smaller advice

- If you are interested in up-to-date performance benchmarks, take a look at www.datalad.org/test_fs_analysis.html[422]. This can help to set expectations and give useful comparisons of file systems or software versions.

- git-annex offers a range of tricks to further improve performance in large datasets. For example, it may be useful to not use a standalone git-annex build, but a native git-annex binary (see this comment[423])

- Status reports in datasets with large amounts of files and/or subdatasets can be expensive. Check out the Gist *Speed up status reports in large datasets* (page 276) for solutions.

16.2 Calculate in greater numbers

When creating and populating datasets yourself it may be easy to monitor the overall size of the dataset and its file number, and introduce subdatasets whenever and where ever necessary. It may not be as straightforward when you are not populating datasets yourself, but when *software* or analyses scripts suddenly dump vast amounts of output. Certain analysis software can create myriads of files. A standard FEAT analysis[428,431]in FSL[429], for example, can easily output several dozens of directories and up to thousands of result files per subject. Maybe your own custom scripts are writing out many files as outputs, too. Regardless of *why* a lot of files are produced by an analyses, if the analysis or software in question runs on a substantially sized input dataset, the results may overwhelm the capacities of a single dataset.

This section demonstrates some tips on how to prevent swamping your datasets with files. If you already accidentally got stuck with an overflowing dataset, checkout section *Fixing up too-large datasets* (page 328) first.

Solution: Subdatasets

To stick to the example of FEAT, here is a quick overview on what this software does: It is modeling neuroimaging data based on general linear modeling (GLM), and creates web page analyses reports, color activation images, time-course plots of data and model, preprocessed intermediate data, images with filtered data, statistical output images, color rendered output images, log files, and many more – in short: A LOT of files. Plenty of these outputs are text-based, but there are also many sizable files. Depending on the type of analysis, not all types of outputs will be relevant. At the end of the analysis, one usually has session-, subject-specific, or aggregated "group" directories with many subdirectories filled with log files, intermediate and preprocessed files, and results for all levels of the analysis.

In such a setup, the output directories (be it on a session/run, subject, or group level) are predictably named, or custom nameable. In order to not flood a single dataset, therefore, one can pre-create appropriate subdatasets of the necessary granularity and have them filled by their analyses. This approach is by no means limited to analyses with certain software, and can be automated. For scripting languages other than Python or shell, standard system calls

---

[422] https://www.datalad.org/test_fs_analysis.html
[423] https://github.com/datalad/datalad/issues/3869#issuecomment-557598390
[428] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FEAT/UserGuide
[431] FEAT is a software tool for model-based fMRI data analysis and part of of FSL[432].
[432] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki
[429] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki

can create output directories as DataLad subdatasets right away, Python scripts can even use DataLad's Python API[433]. Thus, you can create scripts that take care of subdataset creation, or, if you write analysis scripts yourself, you can take care of subdataset creation right in the scripts that are computing and saving your results.

As it is easy to link datasets and operate (e.g., save, clone) across dataset hierarchies, splitting datasets into a hierarchy of datasets does not have many downsides. One substantial disadvantage, though, is that on their own, results in subdirectories don't have meaningful provenance attached. The information about what script or software created them is attached to the superdataset. Should only the subdataset be cloned or inspected, the information on how it was generated is not found.

Solutions without creating subdatasets

It is also possible to scale up without going through the complexities of creating several subdatasets, or tuning your scaling beyond the creation of subdatasets. It involves more thought, or compromising, though. The following section highlights a few caveats to bear in mind if you attempt a big analyses in single-level datasets, and outlines solutions that may not need to involve subdatasets. If you have something to add, please get in touch[430].

Too many files

Caveat: Drown a dataset with too many files.

Examples: The FSL FEAT analysis mentioned in the introduction produces several 100k files, but not all of these files are important. `tsplot/`, for example, is a directory that contains time series plots for various data and results, and may be of little interested for many analyses once general quality control is done.

Solutions:

- Don't put irrelevant files under version control at all: Consider creating a *.gitignore* file with patterns that match files or directories that are of no relevance to you. These files will not be version controlled or saved to your dataset. Section *How to hide content from DataLad* (page 282) can tell you more about this. Be mindful, though: Having too many files in a single directory can still be problematic for your file system. A concrete example: Consider your analyses create log files that are not precious enough to be version controlled. Adding `logs/*` to your `.gitignore` file and saving this change will keep these files out of version control.

- Similarly, you can instruct **datalad run** to save only specific directories or files by specifying them with the `--output` option and executing the command with the `--explicit` flag. This may be more suitable an approach if you know what you want to keep rather than what is irrelevant.

Too many files in Git

Caveat: Drown Git because of configurations.

---

[433] Read more about DataLad's Python API in the first hidden section in *YODA-compliant data analysis projects* (page 143).

[430] https://github.com/datalad-handbook/book/issues/new/

Example: If your dataset is configured with a configuration such as `text2git` or if you have modified your `.gitattributes` file[434] to store files below a certain size of certain types in GIT instead of GIT-ANNEX, an excess of sudden text files can still be overwhelming in terms of total file size. Several thousand, or tens of thousand, text files may still add up to several GB in size even if they are each small in size.

Solutions:

- Add files to git-annex instead of Git: Consider creating custom `largefile` rules for directories that you generate these files in or for patterns that match file names that do not need to be in Git. This way, these files will be put under git-annex's version control. A concrete example: Consider that your analyses output a few thousand text files into all `sub-*/correlations/` directories in your dataset. Appending `sub-*/correlations/*` `annex.largefiles=anything` to `.gitattributes` and saving this change will store all of in the dataset's annex instead of in Git.

- Don't put irrelevant files under version control at all: Consider creating a *.gitignore* file with patterns that match files or directories that are of no relevance to you. These files will not be version controlled or saved to your dataset. Section *How to hide content from DataLad* (page 282) can tell you more about this. Be mindful, though: Having too many files in a single directory can still be problematic for your file system. A concrete example: Consider your analyses create log files that are not precious enough to be version controlled. Adding `logs/*` to your `.gitignore` file and saving this change will keep these files out of version control.

16.3 Fixing up too-large datasets

The previous section highlighted problems of too large monorepos and advised strategies to them prevent them. This section introduces some strategies to clean and fix up datasets that got out of hand size-wise. If there are use cases you would want to see discussed here or propose solutions for, please get in touch[435].

Getting contents out of Git

Let's say you did a **datalad run** with an analysis that put too many files under version control by Git, and you want to see them gone. Sticking to the FSL FEAT analysis example from earlier, you may, for example, want to get rid of every `tsplot` directory, as it contains results that are irrelevant for you.

Note that there is no way to `drop` the files as they are in Git instead of git-annex. Removing the files with plain file system (`rm`, `git rm`) operation also does not shrink your dataset. The files are snapshot and even though they don't exist in the current state of your dataset anymore, they still exist – and thus clutter – your datasets history. In order to *really* get committed files out of Git, you need to rewrite history. And for this you need heavy machinery: git-filter-repo[436,439]. It is a powerful and potentially dangerous tool to rewrite Git history. Treat this tool like a chainsaw.

---

[434] Read up on these configurations in the chapter *Tuning datasets to your needs* (page 112).

[435] https://github.com/datalad-handbook/book/issues/new/

[436] https://github.com/newren/git-filter-repo

[439] Wait, what about `git filter-branch`? Beyond better performance of `git-filter-repo`, Git also discourages the use of `filter-branch` for safety reasons and points to `git-filter-repo` as an alternative. For more background info, see this thread[440].

[440] https://lore.kernel.org/git/CABPp-BEr8LVM+yWTbi76hAq7Moe1hyp2xqxXfgVV4_teh_9skA@mail.gmail.com/

Very helpful for heavy duty tasks, but also life-threatening. The command `git-filter-repo <path-specification> --force` will "filter-out", i.e., remove all files **but the ones specified** in `<path-specification>` from the datasets history. Before you use it, please make sure to read its help page thoroughly.

> 💡 **M16.2 Installing git-filter-repo**
>
> `git-filter-repo` is not part of Git but needs to be installed separately. Its GitHub repository[437] contains more and more detailed instructions, but it is possible to install via PIP (`pip install git-filter-repo`), and available via standard package managers for MacOS and some Linux distributions (mostly rpm-based ones).
>
> ---
> [437] https://github.com/newren/git-filter-repo

The general procedure you should follow is the following:

1. **datalad clone** the repository. This is a safeguard to protect your dataset should something go wrong. The clone you are creating will be your new, cleaned up dataset.

2. **datalad get** all the dataset contents by running `datalad get .` in the clone.

3. `git-filter-repo` what you don't want anymore (see below)

4. Run `git annex unused` and a subsequent `git annex dropunused all` to remove stale file contents that are not referenced anymore.

5. Finally, do some aggressive garbage collection[438] with `git gc --aggressive`

In order to get a hang on the `git-filter-repo` step, consider a directory structure similar to this exemplary run-wise FEAT analysis output structure:

```
$ tree
sub-*/run-*_<task>-<level>.feat
        ├── custom_timing_files
        ├── logs
        ├── reg
        ├── reg_standard
        │   ├── reg
        │   └── stats
        ├── stats
        └── tsplot
```

Each of such `sub-*` directories contains about 3000 files, and the majority of them are irrelevant text files in `tsplot/`. In order to remove them for all subjects and runs from the dataset history, the following command can be used:

```
$ git-filter-repo --path-regex '^sub-[0-9]{2}/run-[0-9]{1}*.feat/tsplot/.*$' --invert-
↪paths --force
```

The `--path-regex` and the regex expression `'^sub-[0-9]{2}/run-[0-9]{1}*.feat/tsplot/.*$'`[441]match all file paths inside of the `tsplot/` directories of all subjects and runs. The option

---

[438] https://git-scm.com/docs/git-gc

[441] Regular expressions can be a pain to comprehend if you're not used to reading them. This one matches paths that start with (`^`) sub- followed by exactly two (`{2}`) numbers that can be between 0 and 9 (`[0-9]`), followed by /run- with exactly one (`{1}`) digit between 0 and 9 (`[0-9]`), followed by zero or more other characters (`*`) until `.feat/tsplot/`, and ending (`$`) with any amount of any character (`.*`). Not exactly easy, but effective. One way to practice reading regular expressions, if you're interested in that, is by playing regex crossword[442].

[442] https://regexcrossword.com/

`--invert-paths` then *inverts* this path specification, and leads to only the files in `tsplot/` to be filtered out. Note that there are also non-regex based path specifications possible, for example with the option `--path-match` or `path-glob`, or with a specification placed in a file. Please see the manual of `git-filter-repo` for more information.

16.4 Summary

If you want to go big, DataLad is a suitable tool and can overcome shortcomings of Git and git-annex, if used correctly. Scaling up involves some thought, and in some instances compromise, though.

- The general mechanism that allows scaling up is nesting datasets. This process can be done by hand or programmatically. Recursive operations ease working across a hierarchy of datasets and create a monorepo-like experience

- Beware of accidentally placing to many (even small) files into Git's version control in a single dataset! `.gitignore` files can keep irrelevant files out of version control, the `explicit` option **datalad run** may be helpful, and custom largefile rules in `.gitattributes` may be necessary to override dataset configurations such as `text2git`.

- Don't consider only the limits of version control software, but also the limits of your file system. Too many files in single directories can become problematic even without version control.

- If things go wrong, it's not all lost. There are ways to clean up your dataset if it ever gets clogged, although they are the software equivalent of a blowtorch and should be handled with care.

Now what can I do with it?

Go big, if you want to. Distribute 80TB of files or more[443], or version control large analyses with minimized performance loss of your version control tools.

---

[443] https://github.com/datalad/datalad-ukbiobank

COMPUTING ON CLUSTERS
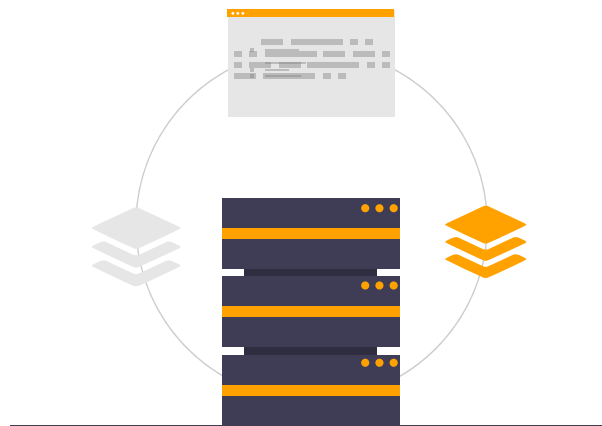
17.1 DataLad on High Throughput or High Performance Compute Clusters

For efficient computing of large analysis, to comply to best computing practices, or to fulfil the requirements that responsible system administrators[444] impose, users may turn to computational clusters such as HIGH-PERFORMANCE COMPUTING (HPC) or HIGH-THROUGHPUT COMPUTING (HTC) infrastructure for data analysis, back-up, or storage.

This chapter is a collection of useful resources and examples that aims to help you get started with DataLad-centric workflows on clusters. We hope to grow this chapter further, so please get in touch[445] if you want to share your use case or seek more advice.

Pointers to content in other chapters

To find out more about centralized storage solutions, you may want to checkout the usecase *Building a scalable data storage for scientific computing* (page 443) or the section *Remote Indexed Archives for dataset storage and backup* (page 294).

---

[444] https://xkcd.com/705/
[445] https://github.com/datalad-handbook/book/issues/new/

DataLad installation on a cluster

Users of a compute cluster generally do not have administrative privileges (sudo rights) and thus can not install software as easily as on their own, private machine. In order to get DataLad and its underlying tools installed, you can either bribe (kindly ask) your system administrator[446,447]or install everything for your own user only following the instructions in the paragraph *Linux-machines with no root access (e.g. HPC systems)* (page 19) of the *installation page* (page 10).

17.2 DataLad-centric analysis with job scheduling and parallel computing

> ⚠️ **This workflow has an update!**
>
> The workflow below is valid and working, but over many months and a few very large scale projects we have improved it with a more flexible and scalable setup. Currently, this work can be found as a comprehensive tutorial and bootstrapping script on GitHub (github.com/psychoinformatics-de/fairly-big-processing-workflow[449]), and a corresponding show case implementation with fMRIprep (github.com/psychoinformatics-de/fairly-big-processing-workflow-tutorial[450]). Also, there is an accompanying preprint with more high-level descriptions of the workflow at www.biorxiv.org/content/10.1101/2021.10.12.464122v1[451]. Its main advantages over the workflow below lie in a distributed (and thus independent) setup of all involved dataset locations; built-in support for two kinds of job schedulers (HTCondor, SLURM); enhanced scalability (tested on 42k datasets of the UK Biobank dataset[452]; and use of REMOTE INDEXED ARCHIVE (RIA) STOREs that provide support for additional security or technical features. Its advised to use the updated workflow over the one below. In the future, this chapter will be updated with an implementation of the updated workflow.
>
> ---
> [449] https://github.com/psychoinformatics-de/fairly-big-processing-workflow
> [450] https://github.com/psychoinformatics-de/fairly-big-processing-workflow-tutorial
> [451] https://www.biorxiv.org/content/10.1101/2021.10.12.464122v1
> [452] https://www.ukbiobank.ac.uk/

There are data analyses that consist of running a handful of scripts on a handful of files. Those analyses can be done in a couple of minutes or hours on your private computer. But there are also analyses that are so large – either in terms of computations, or with regard to the amount of data that they are run on – that it would takes days or even weeks to complete them. The latter type of analyses typically requires a compute cluster, a job scheduler, and parallelization. The question is: How can they become as reproducible and provenance tracked as the simplistic, singular analysis that were showcased in the handbook so far, and that comfortably fitted on a private computer?

> ⚠️ **Reading prerequisite for distributed computing**
>
> It is advised to read the previous chapter *Go big or go home* (page 323) prior to this one

---
[446] https://hsto.org/getpro/habr/post_images/02e/e3b/369/02ee3b369a0326760a160004aca631dc.jpg
[447] You may not need to bribe your system administrator if you are kind to them. Consider frequent gestures of appreciation, or send a geeky T-Shirt for SysAdminDay[448] (the last Friday in July) – Sysadmins do amazing work!
[448] https://en.wikipedia.org/wiki/System_Administrator_Appreciation_Day

This section is a write-up of how DataLad can be used on a scientific computational cluster with a job scheduler for reproducible and FAIR data analyses at scale. It showcases the general principles behind parallel processing of DataLad-centric workflows with containerized pipelines. While this chapter demonstrates specific containerized pipelines and job schedulers, the general setup is generic and could be used with any containerized pipeline and any job scheduling system.

This section lays the groundwork to the next section, a walk-through through a real life example of containerized fMRIprep[453] preprocessing on the eNKI[454] neuroimaging dataset, scheduled with HTCondor[455].

Why job scheduling?

On scientific compute clusters, job scheduling systems such as HTCondor[456] or slurm[457] are used to distribute computational jobs across the available computing infrastructure and manage the overall workload of the cluster. This allows for efficient and fair use of available resources across a group of users, and it brings the potential for highly parallelized computations of jobs and thus vastly faster analyses.

Consider one common way to use a job scheduler: processing all subjects of a dataset independently and as parallel as the current workload of the compute cluster allows – instead of serially "one after the other". In such a setup, each subject-specific analysis becomes a single job, and the job scheduler fits as many jobs as it can on available COMPUTE NODEs. If a large analysis can be split into many independent jobs, using a job scheduler to run them in parallel thus yields great performance advantages in addition to fair compute resource distribution across all users.

> **M17.1 How is a job scheduler used?**
>
> Depending on the job scheduler your system is using, the looks of your typical job scheduling differ, but the general principle is the same.
>
> Typically, a job scheduler is used *non-interactively*, and a *job* (i.e., any command or series of commands you want run) is *submitted* to the scheduler. This submission starts with a "submit" command of the given job scheduler (such as `condor_submit` for HTCondor or `sbatch` for slurm) followed by a command, script, or *batch/submit-file* that contains job definitions and (potentially) compute resource requirements.
>
> The job scheduler takes the submitted jobs, *queues* them up in a central queue, and monitors the available compute resources (i.e., COMPUTE NODEs) of the cluster. As soon as a computational resource is free, it matches a job from the queue to the available resource and computes the job on this node. Usually, a single submission queues up multiple (dozens, hundreds, or thousands of) jobs. If you are interested in a tutorial for HTCondor, checkout the INM-7 HTCondor Tutorial[458].
>
> ---
> [458] https://jugit.fz-juelich.de/inm7/training/htcondor

---

[453] https://fmriprep.readthedocs.io/
[454] http://fcon_1000.projects.nitrc.org/indi/enhanced/
[455] https://research.cs.wisc.edu/htcondor/
[456] https://research.cs.wisc.edu/htcondor/
[457] https://slurm.schedmd.com/overview.html

Where are the difficulties in parallel computing with DataLad?

In order to capture as much provenance as possible, analyses are best ran with a **datalad run** or **datalad containers-run** command, as these commands can capture and link all relevant components of an analysis, starting from code and results to input data and computational environment. Tip: Make use of datalad run's --dry-run option to craft your run-command (see *Dry-running your run call* (page 78))!

But in order to compute parallel jobs with provenance capture, *each individual job* needs to be wrapped in a run command, not only the submission of the jobs to the job scheduler. This requires multiple parallel run commands on the same dataset. But: Multiple simultaneous datalad (containers-)run invocations in the same dataset are problematic.

- Operations carried out during one **run** command can lead to modifications that prevent a second, slightly later run command from being started

- The **datalad save** command at the end of **datalad run** could save modifications that originate from a different job, leading to mis-associated provenance

- A number of *concurrency issues*, unwanted interactions of processes when they run simultaneously, can arise and lead to internal command failures

Some of these problems can be averted by invoking the (containers-)run command with the --explicit[466] flag. This doesn't solve all of the above problems, though, and may not be applicable to the computation at hand – for example because all jobs write to a similar file or the result files are not known beforehand. Below, you can find a complete, largely platform and scheduling-system agnostic containerized analysis workflow that addressed the outlined problems.

Processing FAIRly *and* in parallel – General workflow

> ⚠️ **FAIR and parallel: more than one way to do it**
>
> FAIR *and* parallel processing requires out-of-the-box thinking, and many creative approaches can lead to success. Here is **one** approach that leads to a provenance-tracked, computationally reproducible, and parallel preprocessing workflow, but many more can work. We are eager to hear about yours[459].
>
> ---
> [459] https://github.com/datalad-handbook/book/issues/new/

General setup: The overall setup consists of a data analysis with a containerized pipeline (i.e., a software container that performs a single or a set of analyses). Results will be aggregated into a top-level analysis dataset while the input dataset and a "pipeline" dataset (with a configured software container) exist as subdatasets. The analysis is carried out on a computational cluster that uses a job scheduling system to distribute compute jobs.

The "creative" bits involved in this parallelized processing workflow boil down to the following tricks:

- Individual jobs (for example subject-specific analyses) are computed in **throw-away dataset clones** to avoid unwanted interactions between parallel jobs.

---
[466] To re-read about **datalad run**'s --explicit option, take a look into the section *Clean desk* (page 78).

- Beyond computing in job-specific, temporary locations, individual job results are also saved into uniquely identified BRANCHes to enable simple **pushing back of the results** into the target dataset.

- The jobs constitute a complete DataLad-centric workflow in the form of a simple **bash script**, including dataset build-up and tear-down routines in a throw-away location, result computation, and result publication back to the target dataset. Thus, instead of submitting a datalad run command to the job scheduler, **the job submission is a single script**, and this submission is easily adapted to various job scheduling call formats.

- Right after successful completion of all jobs, the target dataset contains as many BRANCHes as jobs, with each branch containing the results of one job. A manual MERGE aggregates all results into the MASTER branch of the dataset.

The keys to the success of this workflow lie in

- creating it completely *job-scheduling* and *platform agnostic*, such that the workflow can be deployed as a subject/. . . -specific job anywhere, with any job scheduling system, and . . .

- instead of computing job results in the same dataset over all jobs, temporary clones are created to hold individual, job-specific results, and those results are pushed back into the target dataset in the end . . .

- while all dataset components (input data, containerized pipeline) are reusable and the results completely provenance-tracked.

Step-by-Step

To get an idea of the general setup of parallel provenance-tracked computations, consider a *YODA-compliant* (page 136) data analysis dataset. . .

```
$ datalad create parallel_analysis
[INFO   ] Creating a new annex repo at /tmp/parallel_analysis
[INFO   ] Scanning for unlocked files (this may take some time)
create(ok): /tmp/parallel_analysis (dataset)
$ cd parallel_analysis
```

. . . with input data as a subdataset . . .

```
$ datalad clone -d . /path/to/my/rawdata
[INFO   ] Scanning for unlocked files (this may take some time)
install(ok): /tmp/parallel_analysis/rawdata (dataset)
add(ok): /tmp/parallel_analysis/rawdata (file)
add(ok): /tmp/parallel_analysis/.gitmodules (file)
save(ok): /tmp/parallel_analysis (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

. . . and a dataset with a containerized pipeline (for example from the ReproNim container-collection[460,467]) as another subdataset:

---

[460] https://github.com/repronim/containers
[467] The ReproNim container-collection[468] is a DataLad dataset that contains a range of preconfigured containers for neuroimaging.
[468] https://github.com/repronim/containers

```
$ datalad clone -d . https://github.com/ReproNim/containers.git
 [INFO   ] Scanning for unlocked files (this may take some time)
 install(ok): /tmp/parallel_analysis/containers (dataset)
 add(ok): /tmp/parallel_analysis/containers (file)
 add(ok): /tmp/parallel_analysis/.gitmodules (file)
 save(ok): /tmp/parallel_analysis (dataset)
 action summary:
   add (ok: 2)
   install (ok: 1)
   save (ok: 1)
```

> **M17.2 Why do I add the pipeline as a subdataset?**
>
> You could also add and configure the container using `datalad containers-add` to the top-most dataset. This solution makes the container less usable, though. If you have more than one application for a container, keeping it as a standalone dataset can guarantee easier reuse. For an example on how to create such a dataset yourself, please checkout the Findoutmore in *Starting point: Datasets for software and input data* (page 343) in the real-life walk-through in the next section.

The analysis aims to process the `rawdata` with a pipeline from `containers` and collect the outcomes in the toplevel `parallel_analysis` dataset – FAIRly and in parallel, using `datalad containers-run`.

One way to conceptualize the workflow is by taking the perspective of a single compute job. This job consists of whatever you may want to parallelize over. For an arbitrary example, say your raw data contains continuous moisture measurements in the Arctic, taken over the course of 10 years. Each file in your dataset contains the data of a single day. You are interested in a daily aggregate, and are therefore parallelizing across files – each compute job will run an analysis pipeline on one datafile.

> **M17.3 What are common analysis types to parallelize over?**
>
> The key to using a job scheduler and parallelization is to break down an analysis into smaller, loosely coupled computing tasks that can be distributed across a compute cluster. Among common analysis setups that are suitable for parallelization are computations that can be split into several analysis that each run on one subset of the data – such as one (or some) out of many subjects, acquisitions, or files. The large computation "preprocess 200 subjects" can be split into 200 times the job "preprocess 1 subject", for example. In simulation studies, a commonly parallelized task concerns analyses that need to be ran with a range of different parameters, where each parameter configuration can constitute one job.

What you will submit as a job with a job scheduler is not a `datalad containers-run` call, but a shell script that contains all relevant data analysis steps. Using shell[461] as the language for this script is a straight-forward choice as it allows you to script the DataLad workflow just as you would type it into your terminal. Other languages (e.g., using *DataLad's Python API* (page 158) or system calls in languages such as Matlab) would work as well, though.

Building the job:

`datalad (containers-)run` does not support concurrent execution in the *same* dataset clone.

---

[461] https://en.wikipedia.org/wiki/Shell_script

The solution is as easy as it is stubborn: We simply create one throw-away dataset clone for each job.

> **M17.4 how does one create throw-away clones?**
>
> One way to do this are EPHEMERAL CLONEs, an alternative is to make GIT-ANNEX disregard the datasets annex completely using `git annex dead here`. The latter is more appropriate for this context – we could use an ephemeral clone, but that might deposit data of failed jobs at the origin location, if the job runs on a shared filesystem.

Using throw-away clones involves a build-up, result-push, and tear-down routine for each job. It sounds complex and tedious, but this actually works well since datasets are by nature made for such decentralized, collaborative workflows. We treat cluster compute nodes like contributors to the analyses: They clone the analysis dataset hierarchy into a temporary location, run the computation, push the results, and remove their temporary dataset again[469]. The complete routine is done in a single script, which will be submitted as a job. Here, we build the general structure of this script, piece by piece.

The compute job clones the dataset to a unique place, so that it can run a `containers-run` command inside it without interfering with any other job. The first part of the script is therefore to navigate to a unique location, and clone the analysis dataset to it.

> **M17.5 How can I get a unique location?**
>
> On common HTCondor setups, `/tmp` directories in individual jobs are a job-specific local Filesystem that are not shared between jobs – i.e., unique locations! An alternative is to create a unique temporary directory, e.g., with the `mktemp -d` command on Unix systems.

```
# go into unique location
$ cd /tmp
# clone the analysis dataset
$ datalad clone /path/to/parallel_analysis ds
$ cd ds
```

This dataset clone is *temporary*: It will exist over the course of one analysis/job only, but before it is being purged, all of the results it computed will be pushed to the original dataset. This requires a safe-guard: If the original dataset receives the results from the dataset clone, it knows about the clone and its state. In order to protect the results from someone accidentally synchronizing (updating) the dataset from its linked dataset after is has been deleted, the clone should be created as a "trow-away clone" right from the start. By running `git annex dead here`, GIT-ANNEX disregards the clone, preventing the deletion of data in the clone to affect the original dataset.

```
$ git annex dead here
```

The `datalad push` to the original clone location of a dataset needs to be prepared carefully. The job computes *one* result (out of of many results) and saves it, thus creating new data and a new entry with the run-record in the dataset history. But each job is unaware of the results and COMMITs produced by other branches. Should all jobs push back the results to the original place (the MASTER BRANCH of the original dataset), the individual jobs would conflict with each other or, worse, overwrite each other (if you don't have the default push configuration of Git).

---

[469] Clean-up routines can, in the case of common job schedulers, be taken care of by performing everything in compute node specific `/tmp` directories that are wiped clean after job termination.

The general procedure and standard GIT workflow for collaboration, therefore, is to create a change on a different, unique BRANCH, push this different branch, and integrate the changes into the original master branch via a MERGE in the original dataset[470].

In order to do this, prior to executing the analysis, the script will *checkout* a unique new branch in the analysis dataset. The most convenient name for the branch is the Job-ID, an identifier under which the job scheduler runs an individual job. This makes it easy to associate a result (via its branch) with the log, error, or output files that the job scheduler produces[471], and the real-life example will demonstrate these advantages more concretely.

```
# git checkout -b <name> creates a new branch and checks it out
$ git checkout -b "job-$JOBID"
```

Importantly, the $JOB-ID isn't hardcoded into the script but it can be given to the script as an environment or input variable at the time of job submission. The code snippet above uses a bash ENVIRONMENT VARIABLE ($JOBID, as indicated by the all-upper-case variable name with a leading $). It will be defined in the job submission – this is shown and explained in detail in the respective paragraph below.

Next, its time for the **containers-run** command. The invocation will depend on the container and dataset configuration (both of which are demonstrated in the real-life example in the next section), and below, we pretend that the container invocation only needs an input file and an output file. These input file is specified via a bash variables ($inputfile) that will be defined in the script and provided at the time of job submission via command line argument from the job scheduler, and the output file name is based on the input file name.

```
$ datalad containers-run \
    -m "Computing results for $inputfile" \
    --explicit \
    --output "aggregate_${inputfile}" \
    --input "rawdata/$inputfile" \
    -n code/containers/mycontainer \
    '{inputs}' '{outputs}'
```

After the containers-run execution in the script, the results can be pushed back to the dataset SIBLING origin[472]:

```
$ datalad push --to origin
```

Pending a few yet missing safe guards against concurrency issues and the definition of job-specific (environment) variables, such a script can be submitted to any job scheduler with identifiers for input files, output files, and a job ID as identifiers for the branch names. This workflow sketch takes care of everything that needs to be done apart from combining all computed results afterwards.

---

[470] For an analogy, consider a group of software developers: Instead of adding code changes to the main BRANCH of a repository, they develop in their own repository clones and on dedicated, individual feature branches. This allows them to integrate their changes back into the original repository with as little conflict as possible. To find out why a different branch is required to enable easy pushing back to the original dataset, please checkout the explanation on *pushing to non-bare repositories* (page 269) in the section on *How to get help* (page 258).

[471] Job schedulers can commonly produce log, error, and output files and it is advisable to save them for each job. Usually, job schedulers make it convenient to save them with a job-ID as an identifier. An example of this for HTCondor is shown in the Findoutmore in *Job submission* (page 347).

[472] When a dataset is cloned from any location, this original location is by default known as the SIBLING/REMOTE origin to the clone.

Chapter 17. Computing on clusters

> 💡 **M17.6 Fine-tuning: Safe-guard concurrency issues**
>
> An important fine-tuning is missing: Cloning and pushing *can* still run into concurrency issues in the case when one job clones the original dataset while another job is currently pushing its results into this dataset. Therefore, a trick can make sure that no two clone or push commands are executed at *exactly* the same time. This trick uses file locking[462], in particular the tool flock[463], to prevent exactly concurrent processes. This is done by prepending `clone` and `push` commands with `flock --verbose $DSLOCKFILE`, where `$DSLOCKFILE` is a text file placed into `.git/` at the time of job submission, provided via environment variable (see below and the paragraph "Job submission"). This is a non-trivial process, but luckily, you don't need to understand file locking or `flock` in order to follow along – just make sure that you copy the usage of `$DSLOCKFILE` in the script and in the job submission.
>
> _____
>
> [462] https://en.wikipedia.org/wiki/File_locking
> [463] https://www.tutorialspoint.com/unix_system_calls/flock.htm

> 💡 **M17.7 Variable definition**
>
> There are two ways to define variables that a script can use: The first is by defining ENVIRONMENT VARIABLEs, and passing this environment to the compute job. This can be done in the job submission file. To set and pass down the job-ID and a lock file in HTCondor, one can supply the following line in the job submission file:
>
> ```
> environment = "JOBID=$(Cluster).$(Process) DSLOCKFILE=$ENV(PWD)/.git/datalad_lock"
> ```
>
> The second way is via shell script command line arguments. Everything that is given as a command line argument to the script can be accessed in the script in the order of their appearance via $. A script invoked with `bash myscript.sh <inputfile> <parameter> <argument>` can access `inputfile` with `$1`, `parameter` with `$2`, and `<argument>` with `$3`. If the job scheduler takes care of iterating through input file names, the relevant input variable for the simplistic example could thus be defined in the script as follows:
>
> ```
> inputfile=$1
> ```

With fine tuning and variable definitions in place, the only things missing are a SHEBANG at the top of the script, and some shell settings for robust scripting with verbose log files (`set -e -u -x`). Here's how the full general script looks like.

```bash
#!/bin/bash

# fail whenever something is fishy, use -x to get verbose logfiles
set -e -u -x

# we pass arbitrary arguments via job scheduler and can use them as variables
fileid=$1
...


# go into unique location
cd /tmp
# clone the analysis dataset. flock makes sure that this does not interfere
# with another job finishing and pushing results back at the same time
flock --verbose $DSLOCKFILE datalad clone /path/to/parallel_analysis ds
cd ds
```

(continues on next page)

```
# announce the clone to be temporary
git annex dead here
# checkout a unique branch
git checkout -b "job-$JOBID"
# run the job
datalad containers-run \
  -m "Computing data $inputfile" \
  --explicit \
  --output "aggregate_${inputfile}" \
  --input "rawdata/$inputfile" \
  -n code/containers/mycontainer \
  '{inputs}' '{outputs}'
# push, with filelocking as a safe-guard
flock --verbose $DSLOCKFILE datalad push --to origin

# Done - job handler should clean up workspace
```

Its a short script that encapsulates a complete workflow. Think of it as the sequence of necessary DataLad commands you would need to do in order to compute a job. You can save this script into your analysis dataset, e.g., as code/analysis_job.sh, and make it executable (such that it is executed automatically by the program specified in the SHEBANG)using chmod +x code/analysis_job.sh.

Job submission:

Job submission now only boils down to invoking the script for each participant with the relevant command line arguments (e.g., input files for our artificial example) and the necessary environment variables (e.g., the job ID that determines the branch name that is created, and one that points to a lockfile created beforehand once in .git). Job scheduler such as HTCondor can typically do this with automatic variables. They for example have syntax that can identify subject IDs or consecutive file numbers from consistently named directory structure, access the job ID, loop through a predefined list of values or parameters, or use various forms of pattern matching. Examples of this are demonstrated here[464]. Thus, the submit file takes care of defining hundreds or thousands of variables, but can still be lean even though it queues up hundreds or thousands of jobs. Here is a submit file that could be employed:

> 💡 **M17.8 HTCondor submit file**

---

[464] https://jugit.fz-juelich.de/inm7/training/htcondor/-/blob/master/03_define_jobs.md

```
universe       = vanilla
get_env        = True
# resource requirements for each job, determined by
# investigating the demands of a single test job
request_cpus   = 1
request_memory = 20G
request_disk   = 210G


executable     = $ENV(PWD)/code/analysis_job.sh


# the job expects to environment variables for labeling and synchronization
environment = "JOBID=$(Cluster).$(Process) DSLOCKFILE=$ENV(PWD)/.git/datalad_lock"
log     = $ENV(PWD)/../logs/$(Cluster).$(Process).log
output = $ENV(PWD)/../logs/$(Cluster).$(Process).out
error  = $ENV(PWD)/../logs/$(Cluster).$(Process).err
arguments = $(inputfile)
# find all input data, based on the file names in the source dataset.
# The pattern matching below finds all *files* that match the path
# "rawdata/acquisition_*.txt".
# Each relative path to such a file name will become the value of `inputfile`,
# the argument given to the executable (the shell script).
# This will queue as many jobs as file names match the pattern
queue inputfile matching files rawdata/acquisition_*_.txt
```

How would the first few jobs look like that this submit file queues up? It would send out the commands

```
./code/analysis_job.sh rawdata/acquisition_day1year1_.txt
./code/analysis_job.sh rawdata/acquisition_day2year1_.txt
[...]
```

and each of them are send to a compute node with at least 1 CPU, 20GB of RAM and 210GB of disk space. The log, output, and error files are saved under a HTCondor-specific Process and Cluster ID in a log file directory (which would need to be created for HTCondor!). Two environment variables, JOBID (defined from HTCondor-specific Process and Cluster IDs) and DSLOCKFILE (for file locking), will be defined on the compute node.

All it takes to submit is a single condor_submit <submit_file>.

Merging results: Once all jobs are finished, the results lie in individual branches of the original dataset. The only thing left to do now is merging all of these branches into MASTER – and potentially solve any merge conflicts that arise. Usually, merging branches is done using the git merge command with a branch specification. For example, in order to merge one job branch into the MASTER BRANCH, one would need to be on master and run git merge <job branch name>. Given that the parallel job execution could have created thousands of branches, and that each merge would lead to a commit, in order to not inflate the history of the dataset with hundreds of MERGE commits, one can do a single Octopus merges[465] of all branches at once.

> **M17.9 What is an octopus merge?**
>
> Usually a commit that arises from a merge has two *parent* commits: The *first parent* is the branch the merge is being performed from, in the example above, master. The *second parent* is the branch that was merged into the first.

---

[465] https://git-scm.com/docs/git-merge#Documentation/git-merge.txt-octopus

> 💡 However, `git merge` is capable of merging more than two branches simultaneously if more than a single branch name is given to the command. The resulting merge commit has as many parent as were involved in the merge. If a commit has more than two parents, if is affectionately called an "Octopus" merge.
> Octopus merges require merge-conflict-free situations, and will not be carried out whenever manual resolution of conflicts is needed.

The merge command can be assembled quickly. If all result branches were named `job-<JOBID>`, a complete list of branches is obtained with the following command:

```
$ git branch -l | grep 'job-' | tr -d ' '
```

This command line call translates to: "list all branches. Of those branches, show me those that contain `job-`, and remove (`tr -d`) all whitespace." This call can be given to `git merge` as in

```
$ git merge -m "Merge results from job cluster XY" $(git branch -l | grep 'job-' | tr -d
↪' ')
```

Voilà – the results of all provenance-tracked job executions merged into the original dataset. If you are interested in seeing this workflow applied in a real analysis, read on into the next section, *Walkthrough: Parallel ENKI preprocessing with fMRIprep* (page 342).

17.3 Walkthrough: Parallel ENKI preprocessing with fMRIprep

> ⚠️ **This workflow has an update!**
>
> The workflow below is valid and working, but over many months and a few very large scale projects we have improved it with a more flexible and scalable setup. Currently, this work can be found as a comprehensive tutorial and bootstrapping script on GitHub (github.com/psychoinformatics-de/fairly-big-processing-workflow[473]), and a corresponding show case implementation with fMRIprep (github.com/psychoinformatics-de/fairly-big-processing-workflow-tutorial[474]). Also, there is an accompanying preprint with more high-level descriptions of the workflow at www.biorxiv.org/content/10.1101/2021.10.12.464122v1[475]. Its main advantages over the workflow below lie in a distributed (and thus independent) setup of all involved dataset locations; built-in support for two kinds of job schedulers (HTCondor, SLURM); enhanced scalability (tested on 42k datasets of the UK Biobank dataset[476]; and use of REMOTE INDEXED ARCHIVE (RIA) STOREs that provide support for additional security or technical features. Its advised to use the updated workflow over the one below. In the future, this chapter will be updated with an implementation of the updated workflow.
>
> ---
> [473] https://github.com/psychoinformatics-de/fairly-big-processing-workflow
> [474] https://github.com/psychoinformatics-de/fairly-big-processing-workflow-tutorial
> [475] https://www.biorxiv.org/content/10.1101/2021.10.12.464122v1
> [476] https://www.ukbiobank.ac.uk/

The previous section has been an overview on parallel, provenance-tracked computations in DataLad datasets. While the general workflow entails a complete setup, its usually easier to understand it by seeing it applied to a concrete usecase. Its even more informative if that usecase includes some complexities that do not exist in the "picture-perfect" example but are

likely to arise in real life. Therefore, the following walk-through in this section is a write-up of an existing and successfully executed analysis.

The analysis

The analysis goal was standard data preprocessing using fMRIprep[477] on neuroimaging data of 1300 subjects in the eNKI[478] dataset. This computational task is ideal for parallelization: Each subject can be preprocessed individually, each preprocessing takes between 6 and 8 hours per subject, resulting in 1300x7h of serial computing, but only about 7 hours of computing time when executed completely in parallel, and fMRIprep is a containerized pipeline that can be pointed to a specific subject to preprocess.

ENKI was transformed into a DataLad dataset beforehand, and to set up the analysis, the fMRIprep container was placed – with a custom configuration to make it generalizable – into a new dataset called `pipeline`. Both of these datasets, input data and `pipeline` dataset, became subdataset of a data analysis superdataset. In order to associate input data, containerized pipeline, and outputs, the analysis was carried out in a toplevel analysis DataLad dataset and with the **datalad containers-run** command. Finally, as an additional complexity, due to the additional complexity of a large quantity of results, the output was collected in subdatasets.

Starting point: Datasets for software and input data

At the beginning of this endeavour, two important analysis components already exist as DataLad datasets:

1. The input data

2. The containerized pipeline

Following the *YODA principles* (page 136), each of these components is a standalone dataset. While the input dataset creation is straightforwards, some thinking went into the creation of containerized pipeline dataset to set it up in a way that allows it to be installed as a subdataset and invoked from the superdataset. If you are interested in this, find the details in the findout-more below. Also note that there is a large collection of pre-existing container datasets available at github.com/ReproNim/containers[479].

> 💡 **M17.10 pipeline dataset creation**
>
> We start with a dataset (called `pipelines` in this example):
>
> ```
> $ datalad create pipelines
>   [INFO   ] Creating a new annex repo at /data/projects/enki/pipelines
>   create(ok): /data/projects/enki/pipelines (dataset)
> $ cd pipelines
> ```
>
> As one of tools used in fMRIprep's the pipeline, freesurfer[480], requires a license file, this license file needs to be added into the dataset. Only then can this dataset be moved around flexibly and also to different machines. In order to have the license file available right away, it is saved `--to-git` and not annexed[484]:

---

[477] https://fmriprep.readthedocs.io/
[478] http://fcon_1000.projects.nitrc.org/indi/enhanced/
[479] https://github.com/ReproNim/containers

```
$ cp <location/to/fs-license.txt> .
$ datalad save --to-git -m "add freesurfer license file" fs-license.txt
```

Finally, we add a container with the pipeline to the dataset using **datalad containers-add**[485]. The important part is the configuration of the container – it has to be done in a way that makes the container usable in any superdataset the pipeline dataset.

Depending on how the container/pipeline needs to be called, the configuration differs. In the case of an fMRIprep run, we want to be able to invoke the container from a data analysis superdataset. The superdataset contains input data and `pipelines` dataset as subdatasets, and will collect all of the results. Thus, these are arguments we want to supply the invocation with (following fMRIprep's documentation[481]) during a containers-run command:

```
$ datalad containers-run \
[...]
<BIDS_dir> <output_dir> <analysis_level> \
--n_cpus <N> \
--participant-label <ID> \
[...]
```

Note how this list does not include bind-mounts of the necessary directories or of the freesurfer license – this makes the container invocation convenient and easy for any user. Starting an fMRIprep run requires only a `datalad containers-run` with all of the desired fMRIprep options.

This convenience for the user requires that all of the bind-mounts should be taken care of – in a generic way – in the container call specification, though. Here is how this is done:

```
$ datalad containers-add fmriprep \
  --url /data/project/singularity/fmriprep-20.2.0.simg \
  --call-fmt singularity run --cleanenv -B "$PWD" {img} {cmd} --fs-license-file "
↪$PWD/{img_dspath}/freesurfer_license.txt"
```

During a **datalad containers-run** command, the `--call-fmt` specification will be used to call the container. The placeholders `{img}` and `{cmd}` will be replaced with the container (`{img}`) and the command given to `datalad containers-run` (`{cmd}`). Thus, the `--cleanenv` flag as well as bind-mounts are handled prior to the container invocation, and the `--fs-license-file` option with a path to the license file within the container is appended to the command. Bind-mounting the working directory (`-B "$PWD"`) makes sure to bind mount the directory from which the container is being called, which should be the superdataset that contains input data and `pipelines` subdataset. With these bind-mounts, input data and the freesurfer license file within `pipelines` are available in the container.

With such a setup, the `pipelines` dataset can be installed in any dataset and will work out of the box.

---

[480] https://surfer.nmr.mgh.harvard.edu/

[484] If the distinction between annexed and unannexed files is new to you, please read section *Data integrity* (page 85)

[485] Note that this requires the datalad containers extension. Find an overview of all datalad extensions in *DataLad extensions* (page 285).

[481] https://fmriprep.org/en/stable/usage.html

Analysis dataset setup

The size of the input dataset and the nature of preprocessing results with fMRIprep constitute an additional complexity: Based on the amount of input data and test runs of fMRIprep on single subjects, we estimated that the preprocessing results from fMRIprep would encompass several TB in size and about half a million files. This amount of files is too large to be stored in a single dataset, though, and results will therefore need to be split into two result datasets. These will be included as direct subdatasets of the toplevel analysis dataset. This is inconvenient – it separates results (in the result subdatasets) from their provenance (the run-records in the top-level dataset) – but inevitable given the dataset size. A final analysis dataset will consist of the following components:

- input data as a subdataset

- `pipelines` container dataset as a subdataset

- subdatasets to hold the results

Following the benchmarks and tips in the chapter *Go big or go home* (page 323), the amount of files produced by fMRIprep on 1300 subjects requires two datasets to hold them. In this particular computation, following the naming scheme and structure of fMRIpreps output directories, one subdataset is created for the freesurfer[482] results of fMRIprep in a subdataset called `freesurfer`, and one for the minimally preprocessed input data in a subdataset called `fmriprep`.

Here is an overview of the directory structure in the superdataset:

```
superds
├── code                 # directory
│   └── pipelines        # subdataset with fMRIprep
├── fmriprep             # subdataset for results
├── freesurfer           # subdataset for results
└── sourcedata           # subdataset with BIDS-formatted data
    ├── sourcedata       # subdataset with raw data
    ├── sub-A00008326    # directory
    ├── sub-...
```

When running fMRIprep on a smaller set of subjects, or a containerized pipeline that produces fewer files, saving results into subdatasets isn't necessary.

Workflow script

Based on the general principles introduced in the previous section, there is a sketch of the workflow in the `BASH` (shell) script below. It still lacks `fMRIprep` specific fine-tuning – the complete script is shown in the findoutmore afterwards. This initial sketch serves to highlight key differences and adjustments due to the complexity and size of the analysis, explained below and highlighted in the script as well:

- **Getting subdatasets**: The empty result subdatasets wouldn't be installed in the clone automatically – `datalad get -n -r -R1 .` installs all first-level subdatasets so that they are available to be populated with results.

- **recursive throw-away clones**: In the simpler general workflow, we ran `git annex dead here` in the topmost dataset. This dataset contains the results within subdatasets. In order to make them "throw-away" as well, the `git annex dead here` configuration needs to be

---

[482] https://surfer.nmr.mgh.harvard.edu/

applied recursively for all datasets with `git submodule foreach --recursive git annex dead here`.

- **Checkout unique branches in the subdataset**: Since the results will be pushed from the subdatasets, it is in there that unique branches need to be checked out. We're using `git -C <path>` to apply a command in dataset under `path`.

- **Complex container call**: The `containers-run` command is more complex because it supplies all desired fMRIprep arguments.

- **Push the subdatasets only**: We only need to push the results, i.e., there is one push per each subdataset.

```
# everything is running under /tmp inside a compute job,
# /tmp is job-specific local filesystem not shared between jobs
$ cd /tmp

# clone the superdataset with locking
$ flock --verbose $DSLOCKFILE datalad clone /data/project/enki/super ds
$ cd ds

# get first-level subdatasets (-R1 = --recursion-limit 1)
$ datalad get -n -r -R1 .

# make git-annex disregard the clones - they are meant to be thrown away
$ git submodule foreach --recursive git annex dead here

# checkout unique branches (names derived from job IDs) in both subdatasets
# to enable pushing the results without interference from other jobs
# In a setup with no subdatasets, "-C <subds-name>" would be stripped,
# and a new branch would be checked out in the superdataset instead.
$ git -C fmriprep checkout -b "job-$JOBID"
$ git -C freesurfer checkout -b "job-$JOBID"

# call fmriprep with datalad containers-run. Use all relevant fMRIprep
# arguments for your usecase
$ datalad containers-run \
  -m "fMRIprep $subid" \
  --explicit \
  -o freesurfer -o fmriprep \
  -i "$1" \
  -n code/pipelines/fmriprep \
  sourcedata . participant \
  --n_cpus 1 \
  --skip-bids-validation \
  -w .git/tmp/wdir \
  --participant-label "$subid" \
  --random-seed 12345 \
  --skull-strip-fixed-seed \
  --md-only-boilerplate \
  --output-spaces MNI152NLin6Asym \
  --use-aroma \
  --cifti-output

# push back the results
$ flock --verbose $DSLOCKFILE datalad push -d fmriprep --to origin
$ flock --verbose $DSLOCKFILE datalad push -d freesurfer --to origin
# job handler should clean up workspace
```

Just like the general script from the last section, this script can be submitted to any job scheduler – here with a subject ID as a $subid command line variable and a job ID as environment variable as identifiers for the fMRIprep run and branch names. At this point, the workflow misses a tweak that is necessary in fMRIprep to enable re-running computations (the complete file is in *this Findoutmore* (page 348).

> **M17.11 Fine-tuning: Enable re-running**
>
> If you want to make sure that your dataset is set up in a way that you have the ability to rerun a computation quickly, the following fMRIprep-specific consideration is important: If fMRIprep finds preexisting results, it will fail to run. Therefore, all outputs of a job need to be removed before the job is started[486]. We can simply add an attempt to do this in the script (it wouldn't do any harm if there is nothing to be removed):
>
> ```
> (cd fmriprep && rm -rf logs "$subid" "$subid.html" dataset_description.json desc-*.
> ↪tsv)
> (cd freesurfer && rm -rf fsaverage "$subid")
> ```
>
> With this in place, the only things missing are a SHEBANG at the top of the script, and some shell settings for robust scripting with verbose log files (set -e -u -x). You can find the full script with rich comments in *this Findoutmore* (page 348).
>
> ---
> [486] The brackets around the commands are called *command grouping* in bash, and yield a subshell environment: www.gnu.org/software/bash/manual/html_node/Command-Grouping.html[487].

Pending modifications to paths provided in clone locations, the above script and dataset setup is generic enough to be run on different systems and with different job schedulers.

Job submission

Job submission now only boils down to invoking the script for each participant with a participant identifier that determines on which subject the job runs, and setting two environment variables – one the job ID that determines the branch name that is created, and one that points to a lockfile created beforehand once in .git. Job scheduler such as HTCondor have syntax that can identify subject IDs from consistently named directories, for example, and the submit file can thus be lean even though it queues up more than 1000 jobs.

You can find the submit file used in this analyses in *this Findoutmore* (page 350).

All it takes to submit is a single condor_submit <submit_file>.

Merging results

Once all jobs have finished, the results lie in individual branches of the output datasets. In this concrete example, the subdatasets fmriprep and freesurfer will each have more than 1000 branches that hold individual job results. The only thing left to do now is merging all of these branches into MASTER – and potentially solve any merge conflicts that arise. As explained in the previous section, the necessary merging was done with Octopus merges[483] – one in each subdataset (fmriprep and freesurfer).

The merge command was assembled with the trick introduced in the previous section, based on job-ID-named branches. Importantly, this needs to be carried out inside of the subdatasets, i.e.,

---
[483] https://git-scm.com/docs/git-merge#Documentation/git-merge.txt-octopus

> ### 💡 M17.12 See the complete bash script
>
> This script is placed in `code/fmriprep_participant_job`. For technical reasons (render-ing of the handbook), we break it into several blocks of code:
>
> ```bash
> #!/bin/bash
>
> # fail whenever something is fishy, use -x to get verbose logfiles
> set -e -u -x
>
> # we pass in "sourcedata/sub-...", extract subject id from it
> subid=$(basename $1)
>
> # this is all running under /tmp inside a compute job, /tmp is a performant
> # local filesystem
> cd /tmp
> # get the output dataset, which includes the inputs as well
> # flock makes sure that this does not interfere with another job
> # finishing at the same time, and pushing its results back
> # importantly, we clone from the location that we want to push the
> # results too
> flock --verbose $DSLOCKFILE \
>     datalad clone /data/project/enki/super ds
>
> # all following actions are performed in the context of the superdataset
> cd ds
> # obtain all first-level subdatasets:
> # dataset with fmriprep singularity container and pre-configured
> # pipeline call; also get the output dataset to prep them for output
> # consumption, we need to tune them for this particular job, sourcedata
> # important: because we will push additions to the result datasets back
> # at the end of the job, the installation of these result datasets
> # must happen from the location we want to push back too
> datalad get -n -r -R1 .
> # let git-annex know that we do not want to remember any of these clones
> # (we could have used an --ephemeral clone, but that might deposite data
> # of failed jobs at the origin location, if the job runs on a shared
> # filesystem -- let's stay self-contained)
> git submodule foreach --recursive git annex dead here
> ```

```
# checkout new branches in both subdatasets
# this enables us to store the results of this job, and push them back
# without interference from other jobs
git -C fmriprep checkout -b "job-$JOBID"
git -C freesurfer checkout -b "job-$JOBID"
# create workdir for fmriprep inside to simplify singularity call
# PWD will be available in the container
mkdir -p .git/tmp/wdir
# pybids (inside fmriprep) gets angry when it sees dangling symlinks
# of .json files -- wipe them out, spare only those that belong to
# the participant we want to process in this job
find sourcedata -mindepth 2 -name '*.json' -a ! -wholename "$1"'*' -delete

# next one is important to get job-reruns correct. We remove all
# anticipated output, such that fmriprep isn't confused by the presence
# of stale symlinks. Otherwise we would need to obtain and unlock file
# content. But that takes some time, for no reason other than being
# discarded at the end
(cd fmriprep && rm -rf logs "$subid" "$subid.html" dataset_description.json desc-*.
↪tsv)
(cd freesurfer && rm -rf fsaverage "$subid")
```

```
# the meat of the matter, add actual parameterization after --participant-label
datalad containers-run \
  -m "fMRIprep $subid" \
  --explicit \
  -o freesurfer -o fmriprep \
  -i "$1" \
  -n code/pipelines/fmriprep \
  sourcedata . participant \
  --n_cpus 1 \
  --skip-bids-validation \
  -w .git/tmp/wdir \
  --participant-label "$subid" \
  --random-seed 12345 \
  --skull-strip-fixed-seed \
  --md-only-boilerplate \
  --output-spaces MNI152NLin6Asym \
  --use-aroma \
  --cifti-output
# selectively push outputs only
# ignore root dataset, despite recorded changes, needs coordinated
# merge at receiving end
flock --verbose $DSLOCKFILE datalad push -d fmriprep --to origin
flock --verbose $DSLOCKFILE datalad push -d freesurfer --to origin

# job handler should clean up workspace
```

💡 **M17.13 HTCondor submit file**

The following submit file was created and saved in `code/fmriprep_all_participants`.
`submit`:

```
universe      = vanilla
get_env       = True
# resource requirements for each job, determined by
# investigating the demands of a single test job
request_cpus   = 1
request_memory = 20G
request_disk   = 210G


executable     = $ENV(PWD)/code/fmriprep_participant_job


# the job expects to environment variables for labeling and synchronization
environment = "JOBID=$(Cluster).$(Process) DSLOCKFILE=$ENV(PWD)/.git/datalad_lock"
log    = $ENV(PWD)/../logs/$(Cluster).$(Process).log
output = $ENV(PWD)/../logs/$(Cluster).$(Process).out
error  = $ENV(PWD)/../logs/$(Cluster).$(Process).err
arguments = $(subid)
# find all participants, based on the subdirectory names in the source dataset
# each relative path to such a subdirectory with become the value of `subid`
# and another job is queued. Will queue a total number of jobs matching the
# number of matching subdirectories
queue subid matching dirs sourcedata/sub-*
```

within `fmriprep` and `freesurfer`.

```
$ git merge -m "Merge results from job cluster XY" $(git branch -l | grep 'job-' | tr -d
↪' ')
```

Merging with merge conflicts

When attempting an octopus merge like the one above and a merge conflict arises, the merge is
aborted automatically. This is what it looks like in `fmriprep/`, in which all jobs created a slightly
different `CITATION.md` file:

```
$ cd fmriprep
$ git merge -m "Merge results from job cluster 107890" $(git branch -l | grep 'job-' | tr␣
↪-d ' ')
 Fast-forwarding to: job-107890.0
 Trying simple merge with job-107890.1
 Simple merge did not work, trying automatic merge.
 ERROR: logs/CITATION.md: Not merging symbolic link changes.
 fatal: merge program failed
 Automated merge did not work.
 Should not be doing an octopus.
 Merge with strategy octopus failed.
```

This merge conflict is in prinicple helpful – since there are multiple different `CITATION.md` files
in each branch, Git refuses to randomly pick one that it likes to keep, and instead aborts so that
the user can intervene.

💡 **M17.14 How to fix this?**

As the file `CITATION.md` does not contain meaningful changes between jobs, one of the files is kept as a backup (e.g., copied into a temporary location, or brought back to life afterwards with `git cat-file`), then all `CITATION.md` files of all branches deleted prior to the merge, and the back-up `CITATION.md` file is copied and saved into the dataset as a last step.

```
# First, checkout any job branch
$ git checkout job-<insert-number>
# then, copy the file out of the dataset (here, its copied into your home directory)
$ cp logs/CITATION.md ~/CITATION.md
# checkout master again
$ git checkout master
```

Then, remove all CITATION.md files from the last commit. Here is a bash loop that would do exactly that:

```
$ for b in $(git branch -l | grep 'job-' | tr -d ' ');
    do ( git checkout -b m$b $b && git rm logs/CITATION.md && git commit --amend --
↪no-edit ) ;
  done
```

Afterwards, merge the results:

```
$ git merge -m "Merge results from job cluster XY" $(git branch -l | grep 'mjob-' |↪
↪tr -d ' ')
```

Finally, move the back-up file into the dataset:

```
$ mv ~/CITATION.md logs/
$ datalad save -m "Add CITATION file from one job" logs/CITATION.md
```

Merging without merge conflicts

If no merge conflicts arise and the octopus merge is successful, all results are aggregated in the `master` branch. The commit log looks like a work of modern art when visualized with tools such as TIG:
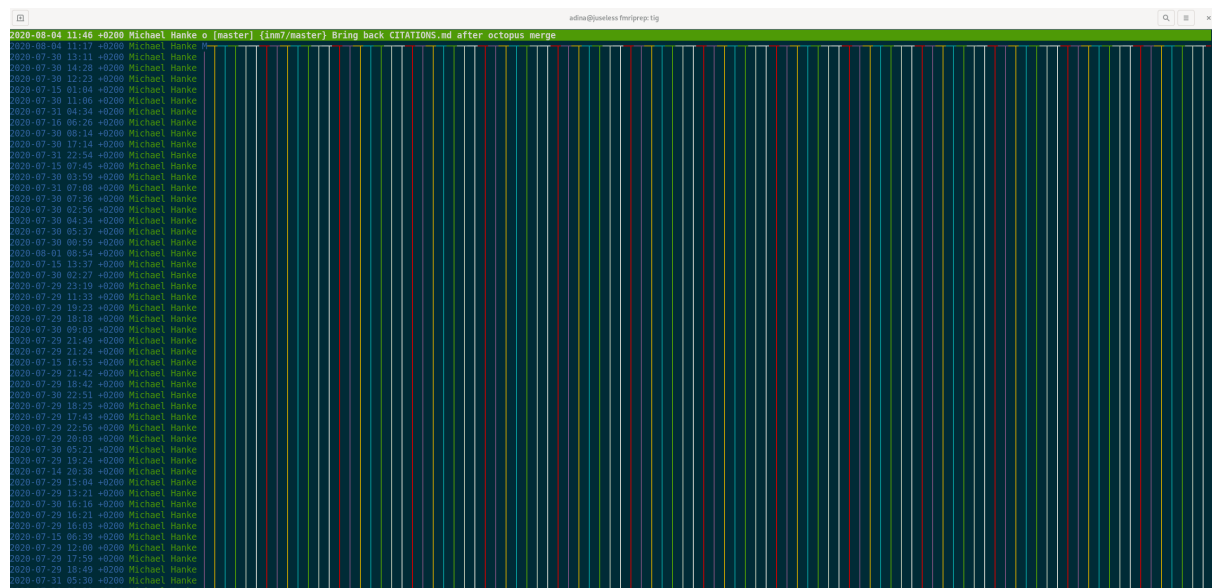
Summary

Once all jobs are computed in parallel and the resulting branches merged, the superdataset is populated with two subdatasets that hold the preprocessing results. Each result contains a machine-readable record of provenance on when, how, and by whom it was computed. From this point, the results in the subdatasets can be used for further analysis, while a record of how they were preprocessed is attached to them.

BETTER LATE THAN NEVER

18.1 Transitioning existing projects into DataLad

Using DataLad offers exciting and useful features that warrant transitioning existing projects into DataLad datasets – and in most cases, transforming your project into one or many DataLad datasets is easy. This sections outlines the basic steps to do so, and offers examples as well as advice and caveats.

Important: Your safety net

Chances are high that you are reading this section of the handbook after you stumbled across DataLad and were intrigued by its features, and you're now looking for a quick way to get going. If you haven't read much of the handbook, but are now planning to DataLad-ify the gigantic project you have been working on for the past months or years, this first paragraph is warning, advice, and a call for safety nets to prevent unexpected misery that can arise from transitioning to a new tool. Because while DataLad *can* do amazing things, you shouldn't blindly trust it to do everything *you* think it can or should do, but gain some familiarity with it.

If you're a DataLad novice, we highly recommend that you read through the *Basics* (page 33) part of the handbook. This part of the book provides you with a solid understanding of DataLad's functionality and a playground to experience working with DataLad. If you're really pressed for time because your dog is sick, your toddler keeps eating your papers and your boss is behind you with a whip, the findoutmore below summarizes the most important sections from the Basics for you to read:

> 💡 **M18.1 The Basics for the impatient**
>
> To get a general idea about DataLad, please read sections *A brief overview of DataLad* (page 2) and *What you really need to know* (page 27) from the introduction (reading time: 15 min).
> To gain a good understanding of some important parts of DataLad, please read chapter *DataLad datasets* (page 34), *DataLad, Run!* (page 59), and *Under the hood: git-annex* (page 83) (reading time: 60 minutes).
> To become confident in using DataLad, sections *How to get help* (page 258), *Miscellaneous file system operations* (page 224) can be very useful. Depending on your aim, *Collaboration* (page 92) (for collaborative workflows), *Third party infrastructure* (page 177) (for data sharing), or *Make the most out of datasets* (page 135) (for data analysis) may contain the relevant background for you.

Prior to transforming your project, regardless of how advanced of a user you are, **we recommend to create a copy of it**. We don't believe there is much that can go wrong from the software-side of things, but data is precious and backups a necessity, so better be safe than sorry.

Step 1: Planning

The first step to DataLad-ify your project is to turn it into one or several nested datasets. Whether you turn a project into a single dataset or several is dependent on the current size of your project and how much you expect it to grow overtime, but also on its contents. You can find guidance on this in paragraph below.

The next step is to save dataset contents. You should take your time and invest thought into this, as this determines the looks and feels of your dataset, in particular the decision on which contents should be saved into GIT or GIT-ANNEX. The section *Data integrity* (page 85) should give you some necessary background information, and the chapter *Tuning datasets to your needs* (page 112) the relevant skills to configure your dataset appropriately. You should consider the size, file type and modification frequency of files in your decisions as well as potential plans to share a dataset with a particular third party infrastructure.

Step 2: Dataset creation

Transforming a directory into a dataset is done with `datalad create --force`. The `-f/--force` option enforces dataset creation in non-empty directories. Consider *applying procedures* (page 126) with `-c <procedure-name>` to apply configurations that suit your use case.

> 💡 **M18.2 What if my directory is already a Git repository?**
>
> If you want to transform a Git repository to a DataLad dataset, a `datalad create -f` is the way to go, too, and completely safe. Your Git history will stay intact and will not be tampered with.

If you want to transform a series of nested directories into nested datasets, continue with `datalad create -f` commands in all further subdirectories.

> 💡 **M18.3 One or many datasets?**
>
> In deciding how many datasets you need, try to follow the benchmarks in chapter *Go big or go home* (page 323) and the yoda principles in section *YODA: Best practices for data analyses in a dataset* (page 136). Two simple questions can help you make a decision:
> 1. Do you have independently reusable components in your directory, for example data from several studies, or data and code/results? If yes, make each individual component a dataset.
> 2. How large is each individual component? If it exceeds 100k files, split it up into smaller datasets. The decision on where to place subdataset boundaries can be guided by the existing directory structure or by common access patterns, for example based on data type (raw, processed, . . .) or subject association. One straightforward organization may be a top-level superdataset and subject-specific subdatasets, mimicking the structure chosen in the use case *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433).

You can automate this with BASH loops, if you want.

> 💡 **M18.4 Example bash loops**
>
> Consider a directory structure that follows a naming standard such as BIDS[488]:
>
> ```
> # create a mock-directory structure:
> $ mkdir -p study/sub-0{1,2,3,4,5}/{anat,func}
> $ tree study
> study
> ├── sub-01
> │   ├── anat
> │   └── func
> ├── sub-02
> │   ├── anat
> │   └── func
> ├── sub-03
> │   ├── anat
> │   └── func
> ├── sub-04
> │   ├── anat
> │   └── func
> └── sub-05
>     ├── anat
>     └── func
> ```
>
> Consider further that you have transformed the toplevel `study` directory into a dataset and now want to transform all `sub-*` directories into further subdatasets, registered in `study`. Here is a line that would do this for the example above:
>
> ```
> $ for dir in study/sub-0{1,2,3,4,5}; do datalad -C $dir create -d^. --force .; done
> ```
>
> ---
> [488] https://bids.neuroimaging.io/

Step 3: Saving dataset contents

Any existing content in your newly created dataset(s) still needs to be saved into its dataset at this point (unless it was already under version control with Git). This can be done with the

datalad save command – either "in one go" using a plain datalad save (saves all untracked files and modifications to a dataset – by default into the dataset annex), or step-by-step by attaching paths to the save command. Make sure to run **datalad status** frequently.

> **M18.5 Save things to Git or to git-annex?**
>
> By default, all dataset contents are saved into GIT-ANNEX. Depending on your data and use case, this may or may not be useful for all files. Here are a few things to keep in mind:
> - large files, in particular binary files should almost always go into GIT-ANNEX. If you have pure data dataset made up of large files, put it into the dataset annex.
> - small files, especially if they are text files and undergo frequent modifications (e.g., code, manuscripts, notes) are best put under version control by GIT.
> - If you plan to publish a dataset to a repository hosting site without annex support such as GITHUB or GITLAB, and do not intend to set up third party storage for annexed contents, be aware that only contents placed in Git will be available to others after cloning your repository. At the same time, be mindful of file size limits the services impose. The largest file size GitHub allows is 100MB – a dataset with files exceeding 100MB in size in Git will be rejected by GitHub. GIN is an alternative hosting service with annex support, and the Open Science Framework (OSF)[489] may also be a suitable option to share datasets including their annexed files.
>
> You can find guidance on how to create configurations for your dataset (which need to be in place and saved prior to saving contents!) in the chapter *Tuning datasets to your needs* (page 112), in particular section *More on DIY configurations* (page 117).
>
> ---
> [489] https://readthedocs.org/projects/datalad-osf/

> **Create desired subdatasets first**
>
> Be mindful during saving if you have a directory that should hold more, yet uncreated datasets down its hierarchy, as a plain datalad save will save *all* files and directories to the dataset! Its best to first create all subdatasets, and only then save their contents.

If you are operating in a hierarchy of datasets, running a recursive save from the top-most dataset (datalad save -r) will save you time: All contents are saved to their respective datasets, all subdatasets are registered to their respective superdatasets.

Step 4: Rerunning analyses reproducibly

If you are transforming a complete data analysis into a dataset, you may also want to rerun any computation with DataLad's run commands. You can compose any **datalad run** or **datalad containers-run**[490] command to recreate and capture your previous analysis. Make sure to specify your previous results as --output in order to unlock them[491].

---

[490] Prior to using a software container, install the *datalad-containers* (page 285) extension and add the container with the **datalad containers-add** command. You can find a concrete data analysis example with datalad-containers in the section *Computational reproducibility with software containers* (page 166).

[491] If you are unfamiliar with datalad run, please work through chapter *DataLad, Run!* (page 59) first.

Summary

Existing projects and analysis can be DataLad-ified with a few standard commands. Be mindful about dataset sizes and whether you save contents into Git or git-annex, though, as these choices could potentially spoil your DataLad experience. The sections *Miscellaneous file system operations* (page 224) and *Fixing up too-large datasets* (page 328) can help you to undo unwanted changes, but its better to do things right instead of having to fix them up. If you can, read up on the DataLad Basics to understand what you are doing, and create a backup in case things go not as planned in your first attempts.

SPECIAL PURPOSE SHOWROOMS
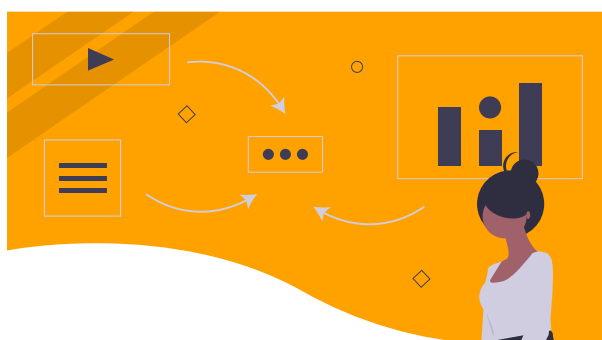
19.1 Reproducible machine learning analyses: DataLad as DVC

Machine learning analyses are complex: Beyond data preparation and general scripting, they typically consist of training and optimizing several different machine learning models and comparing them based on performance metrics. This complexity can jeopardize reproducibility – it is hard to remember or figure out which model was trained on which version of what data and which has been the ideal optimization. But just like any data analysis project, machine learning projects can become easier to understand and reproduce if they are intuitively structured, appropriately version controlled, and if analysis executions are captured with enough (ideally machine-readable and re-executable) provenance.

DataLad provides the functionality to achieve this, and *previous* (page 136) *sections* (page 166) have given some demonstrations on how to do it. But in the context of machine learning analyses, other domain-specific tools and workflows exist, too. One of the most well-known is DVC (Data Version Control)[492], a "version control system for machine learning projects". This section compares the two tools and demonstrates workflows for data versioning, data sharing, and analysis execution[493] in the context of a machine learning project with DVC and DataLad. While they share a number of similarities and goals, their respective workflows are quite distinct.

The workflows showcased here are based on a DVC tutorial[494]. This tutorial consists of the following steps:

- A data set with pictures of 10 classes of objects (Imagenette[495]) is version controlled with DVC

---

[492] https://dvc.org/
[493] https://realpython.com/python-data-version-control/
[494] https://realpython.com/python-data-version-control/
[495] https://github.com/fastai/imagenette

- the data is pushed to a "storage remote" on a local path

- the data are analyzed using various ML models in DVC pipelines

This handbook section demonstrates how DataLad could be used as an alternative to DVC. We demonstrate each step with DVC according to their tutorial, and then recreate a corresponding DataLad workflow. The usecase *DataLad for reproducible machine-learning analyses* (page 454) demonstrates a similar analysis in a completely DataLad-centric fashion. If you want to, you can code along, or simply read through the presentation of DVC and DataLad commands. Some familiarity with DataLad can be helpful, but if you have never used DataLad, footnotes in each section can point you relevant chapters for more insights on a command or concept. If you have never used DVC, its technical docs[496] or collection of third-party tutorials[497] can answer further questions.

If you are not a Git user

DVC relies heavily on Git workflows. Understanding the DVC workflows requires a solid understanding of BRANCHes, Git's concepts of Working tree, Index ("Staging Area"), and Repository[498], and some basic Git commands such as add, commit, and checkout. The Turing Way[499] has an excellent chapter on version control with Git[500] if you want to catch up on those basics first.

---

[496] https://dvc.org/doc/command-reference
[497] https://github.com/iterative/dvc.org/issues/1749
[498] https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository
[499] https://the-turing-way.netlify.app/welcome.html
[500] https://the-turing-way.netlify.app/reproducible-research/vcs.html

> ### G19.1 Terminology
>
> Be mindful: DVC (as DataLad) comes with a range of commands and concepts that have the same names, but differ in functionality to their Git namesake. Make sure to read the DVC documentation[501] for each command to get more information on what it does.
>
> ───────────────────
> [501] https://dvc.org/doc/command-reference

> ### ⚠ Running this tutorial requires DataLad version 0.13.4 or higher
>
> Running this tutorial requires DataLad version 0.13.4 or higher

Setup

The DVC tutorial[502] comes with a pre-made repository that is structured for DVC machine learning analyses. If you want to code along, the repository[503] needs to be FORKed (requires a GitHub account) and cloned from your own fork[525].

```
### DVC
# please clone this repository from your own fork when coding along
$ git clone https://github.com/datalad-handbook/data-version-control DVC
Cloning into 'DVC'...
```

The resulting Git repository is already pre-structured in a way that aids DVC ML analyses: It has the directories `model` and `metrics`, and a set of Python scripts for a machine learning analysis in `src/`.

```
### DVC
$ tree DVC
DVC
├── data
│   ├── prepared
│   └── raw
├── LICENSE
├── metrics
├── model
├── README.md
└── src
    ├── evaluate.py
    ├── prepare.py
    └── train.py

6 directories, 5 files
```

For a comparison, we will recreate a similarly structured DataLad dataset. For greater compliance with DataLad's *YODA principles* (page 136), the dataset structure will differ marginally in that scripts will be kept in `code/` instead of `src/`. We create the dataset with two configurations, yoda and `text2git`[527].

───────────────────

[502] https://realpython.com/python-data-version-control/
[503] https://github.com/datalad-handbook/data-version-control.git
[525] Instructions on FORKing and cloning the repo are in the README of the repository: github.com/realpython/data-version-control[526].
[526] https://github.com/realpython/data-version-control
[527] The two procedures provide the dataset with useful structures and configurations for its purpose: yoda creates

```
### DVC-DataLad
$ datalad create -c text2git -c yoda DVC-DataLad
$ cd DVC-DataLad
$ mkdir -p data/{raw,prepared} model metrics
[INFO] Creating a new annex repo at /home/me/DVCvsDL/DVC-DataLad
[INFO] Running procedure cfg_text2git
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
[INFO] Running procedure cfg_yoda
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
create(ok): /home/me/DVCvsDL/DVC-DataLad (dataset)
```

Afterwards, we make sure to get the same scripts.

```
### DVC-DataLad
# get the scripts
$ datalad download-url -m "download scripts for ML analysis" \
  https://raw.githubusercontent.com/datalad-handbook/data-version-control/master/src/
↪{train,prepare,evaluate}.py \
  -O 'code/'
[INFO] Downloading 'https://raw.githubusercontent.com/datalad-handbook/data-version-
↪control/master/src/train.py' into '/home/me/DVCvsDL/DVC-DataLad/code/'
download_url(ok): /home/me/DVCvsDL/DVC-DataLad/code/train.py (file)
[INFO] Downloading 'https://raw.githubusercontent.com/datalad-handbook/data-version-
↪control/master/src/prepare.py' into '/home/me/DVCvsDL/DVC-DataLad/code/'
download_url(ok): /home/me/DVCvsDL/DVC-DataLad/code/prepare.py (file)
[INFO] Downloading 'https://raw.githubusercontent.com/datalad-handbook/data-version-
↪control/master/src/evaluate.py' into '/home/me/DVCvsDL/DVC-DataLad/code/'
download_url(ok): /home/me/DVCvsDL/DVC-DataLad/code/evaluate.py (file)
add(ok): code/evaluate.py (file)
add(ok): code/prepare.py (file)
add(ok): code/train.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  download_url (ok: 3)
  save (ok: 1)
```

Here's the final directory structure:

```
### DVC-DataLad
$ tree
.
├── CHANGELOG.md
├── code
│   ├── evaluate.py
│   ├── prepare.py
│   ├── README.md
│   └── train.py
├── data
│   ├── prepared
```

(continues on next page)

---

a dataset structure with a code directory and makes sure that everything kept in code will be committed to Git (thus allowing for direct sharing of code). text2git makes sure that any other text file in the dataset will be stored in Git as well. The sections *Data safety* (page 83) and *The YODA procedure* (page 142) explain the two configurations in detail.

```
│       └── raw
├── metrics
├── model
└── README.md
```

6 directories, 6 files

> 💡 **M19.1 Required software for coding along**
>
> In order to code along, DVC[504], scikit-learn[505], scikit-image[506], pandas[507], and numpy[508] are required. All tools are available via pip[509] or conda[510]. We recommend to install them in a virtual environment[511] – the DVC tutorial has step-by-step instructions[512].
>
> ---
>
> [504] https://dvc.org/doc/install
> [505] https://scikit-learn.org/stable/
> [506] https://scikit-image.org/
> [507] https://pandas.pydata.org/
> [508] https://numpy.org/
> [509] https://pypi.org/project/pip/
> [510] https://docs.conda.io/en/latest/
> [511] https://realpython.com/python-data-version-control/#set-up-your-working-environment
> [512] https://realpython.com/python-data-version-control/#set-up-your-working-environment

Version controlling data

In the first part of the tutorial, the directory tree will be populated with data that should be version controlled.

Although the implementation of version control for (large) data is very different between DataLad and DVC, the underlying concept is very similar: (Large) data is stored outside of GIT – GIT only tracks information on where this data can be found.

In DataLad datasets, (large) data is handled by GIT-ANNEX. Data content is hashed[513] and only the hash (represented as the original file name) is stored in Git[528]. Actual data is stored in the ANNEX of the dataset, and annexed data can be transferred from and to a large number of storage solutions[514] using either DataLad or git-annex commands. Information on where data is available from is *stored in an internal representation of git-annex* (page 487).

In DVC repositories, (large) data is also supposed to be stored in external remotes such as Google Drive. For internal representation of where files are available from, DVC uses one `.dvc` text file for each data file or directory given to DVC. The `.dvc` files contain information on the path to the data in the repository, where the associated data file is available from, and a hash, and those files should be committed to GIT.

DVC workflow

Prior to adding and version controlling data, a "DVC project" needs to be initialized in the Git repository:

---

[513] https://en.wikipedia.org/wiki/Hash_function
[528] To re-read about how GIT-ANNEX handles versioning of (large) files, go back to section *Data integrity* (page 85).
[514] https://git-annex.branchable.com/special_remotes/

```
### DVC
$ cd ../DVC
$ dvc init
```

```
You can now commit the changes to git.
```

```
+---------------------------------------------------------------+
|                                                               |
|        DVC has enabled anonymous aggregate usage analytics.   |
|     Read the analytics documentation (and how to opt-out) here: |
|              <https://dvc.org/doc/user-guide/analytics>       |
|                                                               |
+---------------------------------------------------------------+
```

```
What's next?
------------
- Check out the documentation: <https://dvc.org/doc>
- Get help and share ideas: <https://dvc.org/chat>
- Star us on GitHub: <https://github.com/iterative/dvc>
```

This populates the repository with a range of staged[515] files – most of them are internal directories and files for DVC's configuration.

```
### DVC
$ git status
On branch master
Your branch is up to date with 'github/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .dvc/.gitignore
        new file:   .dvc/config
        new file:   .dvc/plots/confusion.json
        new file:   .dvc/plots/confusion_normalized.json
        new file:   .dvc/plots/default.json
        new file:   .dvc/plots/linear.json
        new file:   .dvc/plots/scatter.json
        new file:   .dvc/plots/smooth.json
        new file:   .dvcignore
```

As they are only *staged* but not *committed,* we need to commit them (into Git):

```
### DVC
$ git commit -m "initialize dvc"
[master 38729a0] initialize dvc
 9 files changed, 515 insertions(+)
 create mode 100644 .dvc/.gitignore
 create mode 100644 .dvc/config
 create mode 100644 .dvc/plots/confusion.json
 create mode 100644 .dvc/plots/confusion_normalized.json
 create mode 100644 .dvc/plots/default.json
 create mode 100644 .dvc/plots/linear.json
 create mode 100644 .dvc/plots/scatter.json
 create mode 100644 .dvc/plots/smooth.json
 create mode 100644 .dvcignore
```

---

[515] https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

The DVC project is now ready to version control data. In the tutorial, data comes from the "Imagenette" dataset. This data is available from an Amazon S3 bucket[516] as a compressed tarball, but to keep the download fast, there is a smaller two-category version of it on the OPEN SCIENCE FRAMEWORK (OSF). We'll download it and extract it into the data/raw/ directory of the repository.

```
### DVC
# download the data
$ wget -q https://osf.io/d6qbz/download -O imagenette2-160.tgz
# extract it
$ tar -xzf imagenette2-160.tgz
# move it into the directories
$ mv train data/raw/
$ mv val data/raw/
# remove the archive
$ rm -rf imagenette2-160.tgz
```

The data directories in data/raw are then version controlled with the **dvc add** command that can place files or complete directories under version control by DVC.

```
### DVC
$ dvc add data/raw/train
$ dvc add data/raw/val
```

```
To track the changes with git, run:

        git add data/raw/.gitignore data/raw/train.dvc
```

```
To track the changes with git, run:

        git add data/raw/.gitignore data/raw/val.dvc
```

Here is what this command has accomplished: The data files were copied into a *cache* in .dvc/ cache (a non-human readable directory structure based on hashes similar to *.git/annex/objects* used by *git-annex*), data file names were added to a .gitignore[529] file to become invisible to Git, and two .dvc files, train.dvc and val.dvc, were created[530]. **git status** shows these changes:

```
### DVC
$ git status
On branch master
Your branch is ahead of 'github/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   data/raw/.gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

<div align="right">(continues on next page)</div>

---

[516] https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.tgz

[529] You can read more about .gitignore files in the section *How to hide content from DataLad* (page 282)

[530] If you are curious about why data is duplicated in a cache or why the paths to the data are placed into a .gitignore file, this section in the DVC tutorial[531] has more insights on the internals of this process.

[531] https://realpython.com/python-data-version-control/#tracking-files

```
        data/raw/train.dvc
        data/raw/val.dvc
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

In order to complete the version control workflow, Git needs to know about the `.dvc` files, and forget about the data directories. For this, the modified `.gitignore` file and the untracked `.dvc` files need to be added to Git:

```
### DVC
$ git add --all
```

Finally, we commit.

```
### DVC
$ git commit -m "control data with DVC"
[master baac3ef] control data with DVC
 3 files changed, 12 insertions(+)
 create mode 100644 data/raw/train.dvc
 create mode 100644 data/raw/val.dvc
```

The data is now version controlled with DVC.

> **M19.2 How does DVC represent modifications to data?**
>
> When adding data directories, they (i.e., the complete directory) are hashed, and this hash is stored in the respective `.dvc` file. If any file in the directory changes, this hash would change, and the **dvc status** command would report the directory to be "changed". To demonstrate this, we pretend to accidentally delete a single file:
>
> ```
> # if one or more files in the val/ data changes, dvc status reports a change
> $ dvc status
> data/raw/val.dvc:
>     changed outs:
>         modified:           data/raw/val
> ```
>
> **Important**: Detecting a data modification **requires** the **dvc status** command – **git status** will not be able to detect changes as this directory as it is git-ignored!

DataLad workflow

DataLad has means to get data or data archives from web sources and store this availability information within GIT-ANNEX. This has several advantages: For one, the original S3 bucket is known and stored as a location to re-retrieve the data from. This enables reliable data access for yourself and others that you share the dataset with. Beyond this, the data is also automatically extracted and saved, and thus put under version control. Note that this strays slightly from DataLad's *YODA principles* (page 136) in a DataLad-centric workflow, where data should become a standalone, reusable dataset that would be linked as a subdataset into a study/analysis specific dataset. Here, we stick to the project organization of DVC though.

```
### DVC-DataLad
$ cd ../DVC-DataLad
```

```
$ datalad download-url \
  --archive \
  --message "Download Imagenette dataset" \
  https://osf.io/d6qbz/download \
  -O 'data/raw/'
[INFO] Downloading 'https://osf.io/d6qbz/download' into '/home/me/DVCvsDL/DVC-DataLad/
↪data/raw/'
download_url(ok): /home/me/DVCvsDL/DVC-DataLad/data/raw/imagenette2-160.tgz (file)
add(ok): data/raw/imagenette2-160.tgz (file)
save(ok): . (dataset)
[INFO] Adding content of the archive /home/me/DVCvsDL/DVC-DataLad/data/raw/imagenette2-
↪160.tgz into annex AnnexRepo(/home/me/DVCvsDL/DVC-DataLad)
[INFO] Initiating special remote datalad-archives
[INFO] Finished adding /home/me/DVCvsDL/DVC-DataLad/data/raw/imagenette2-160.tgz: Files␣
↪processed: 2701, renamed: 2701, +annex: 2701
[INFO] Finished extraction
add-archive-content(ok): /home/me/DVCvsDL/DVC-DataLad (dataset)
action summary:
  add (ok: 1)
  add-archive-content (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
```

At this point, the data is already version controlled[532], but the directory structure doesn't resemble that of the DVC dataset yet – the extracted directory adds one unnecessary directory layer:

```
$ tree
.
├── code
│   └── [...]
├── data
│   └── raw
│       ├── train
│       │   ├─[...]
│       └── val
│           ├── [...]
├── metrics
└── model

29 directories
```

To make the scripts work, we move the raw data up one level. This move needs to be saved.

> **M19.3 How does DataLad represent modifications to data?**
>
> As DataLad always tracks files individually, `datalad status` (or, alternatively, `git status` or `git annex status`) will show modifications on the level of individual files:

---

[532] The sections *Populate a dataset* (page 37) and *Modify content* (page 43) introduce the concepts of saving and modifying files in DataLad datasets.

```
$ datalad status
  deleted: /home/me/DVCvsDL/DVC-DataLad/data/raw/val/n01440764/n01440764_12021.JPEG⌣
↪(symlink)

$ git status
  On branch master
  Your branch is ahead of 'origin/master' by 2 commits.
    (use "git push" to publish your local commits)

  Changes not staged for commit:
    (use "git add/rm <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
      deleted:    data/raw/val/n01440764/n01440764_12021.JPEG

$ git annex status
  D data/raw/val/n01440764/n01440764_12021.JPEG
```

Sharing data

In the second part of the tutorial, the versioned data is transferred to a local directory to demonstrate data sharing.

The general mechanisms of DVC and DataLad data sharing are similar: (Large) data files are kept somewhere where potentially large files can be stored. They can be retrieved on demand as the location information is stored in Git. DVC uses the term "data remote" to refer to external storage locations for (large) data, whereas DataLad would refer to them as (storage-) SIBLINGs.

Both DVC and DataLad support a range of hosting solutions, from local paths and SSH servers to providers such as S3 or GDrive. For DVC, every supported remote is pre-implemented, which restricts the number of available services (a list is here[517]), but results in a convenient, streamlined procedure for adding remotes based on URL schemes. DataLad, largely thanks to "external special remotes" mechanism of git-annex, has more storage options (in addition for example *DropBox* (page 177), the Open Science Framework (OSF)[518], *Git LFS* (page 207), *Figshare* (page 215), *GIN* (page 208), or *RIA stores* (page 294)), but depending on selected storage provider, the procedure to add a sibling may differ. In addition, DataLad is able to store complete datasets (annexed data *and* Git repository) in certain services (e.g., OSF, GIN, GitHub if used with GitLFS, Dropbox, . . .), enabling a clone from for example Google Drive, and while DVC can never keep data in Git repository hosting services, DataLad can do this if the hosting service supports hosting annexed data (default on GIN and possible with GITHUB, GITLAB or BITBUCKET if used with GitLFS[519]).

DVC workflow

Step 1: Set up a remote

The DVC tutorial[520] demonstrates data sharing via a local data remote[533]. As a first step, there

---

[517] https://dvc.org/doc/command-reference/remote/add
[518] http://docs.datalad.org/projects/osf/en/latest/
[519] https://git-lfs.github.com/
[520] https://realpython.com/python-data-version-control
[533] A similar procedure for sharing data on a local file system for DataLad is shown in the chapter *Looking without touching* (page 92).

---

needs to exist a directory to use as a remote, so we will create a new directory:

```
### DVC
# go back to DVC (we were in DVC-Datalad)
$ cd ../DVC
# create a directory somewhere else
$ mkdir ../dvc-remote
```

Afterwards, the new, empty directory can be added as a data remote using **dvc remote add**. The -d option sets it as the default remote, which simplifies pushing later on:

```
### DVC
$ dvc remote add -d remote_storage ../dvc_remote
Setting 'remote_storage' as a default remote.
```

The location of the remote is written into a config file:

```
### DVC
$ cat .dvc/config
[core]
    remote = remote_storage
['remote "remote_storage"']
    url = ../../dvc_remote
```

Note that dvc remote add only *modifies* the config file, and it still needs to be added and committed to Git:

```
### DVC
$ git status
On branch master
Your branch is ahead of 'github/master' by 2 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .dvc/config

no changes added to commit (use "git add" and/or "git commit -a")
```

```
### DVC
$ git add .dvc/config
$ git commit -m "add local remote"
[master 8f24d9f] add local remote
 1 file changed, 4 insertions(+)
```

> **G19.2 Remotes**
>
> The DVC and Git concepts of a "remote" are related, but not identical. Therefore, DVC remotes are invisible to **git remote**, and likewise, Git REMOTEs are invisible to the **dvc remote list** command.

Step 2: Push data to the remote

Once the remote is set up, the data that is managed by DVC can be pushed from the *cache* of the project to the remote. During this operation, all data for which .dvc files exist will be copied from .dvc/cache to the remote storage.

```
### DVC
$ dvc push
2703 files pushed
```

Step 3: Push Git history

At this point, all changes that were committed to GIT (such as the .dvc files) still need to be pushed to a Git repository hosting service.

```
### DVC
# this will only work if you have cloned from your own fork
$ git push origin master
To /home/me/pushes/data-version-control
 * [new branch]      master -> master
```

Step 4: Data retrieval

In DVC projects, there are several ways to retrieve data into its original location or the project cache. In order to demonstrate this, we start by deleting a data directory (in its original location, data/raw/val/).

```
### DVC
$ rm -rf data/raw/val
```

> **G19.3 Status**
>
> Do note that this deletion would not be detected by `git status` – you have to use `dvc status` instead.

At this point, a copy of the data still resides in the cache of the repository. These data are copied back to val/ with the `dvc checkout` command:

```
### DVC
$ dvc checkout data/raw/val.dvc
A       data/raw/val/
```

If the cache of the repository would be empty, the data can be re-retrieved into the cache from the data remote. To demonstrate this, let's look at a repository with an empty cache by cloning this repository from GitHub into a new location.

```
### DVC
# clone the repo into a new location for demonstration purposes:
$ cd ../
$ git clone https://github.com/datalad-handbook/data-version-control DVC-2
Cloning into 'DVC-2'...
done.
```

Retrieving the data from the data remote to repopulate the cache is done with the `dvc fetch` command:

```
### DVC
$ cd DVC-2
$ dvc fetch data/raw/val.dvc
789 files fetched
```

Afterwards, another `dvc checkout` will copy the files from the cache back to val/. Alternatively,

---

the command **dvc pull** performs fetch (get data into the cache) and checkout (copy data from the cache to its original location) in a single command.

Unless DVC is used on a small subset of file systems (trfs, XFS, OCFS2, or APFS), copying data between its original location and the cache is the default. This results in a "built-in data duplication" on most current file systems[534]. An alternative is to switch from copies to SYMLINKs (as done by GIT-ANNEX) or hardlinks[521].

DataLad workflow

Because the S3 bucket of the raw data is known and stored in the dataset, it strictly speaking isn't necessary to create a storage sibling to push the data to – DataLad already treats the original S3 bucket as storage. Currently, the dataset can thus be shared via GITHUB or similar hosting services, and the data can be retrieved using **datalad get**.

> **M19.4 Really?**
>
> Sure. Let's demonstrate this. First, we create a sibling on GitHub for this dataset and push its contents to the sibling:
>
> ```
> ### DVC-DataLad
> $ cd ../DVC-DataLad
> $ datalad create-sibling-github DVC-DataLad --github-organization datalad-handbook
> [INFO   ] Successfully obtained information about organization datalad-handbook␣
> ↪using UserPassword(name='github', url='https://github.com/login') credential
>  .: github(-) [https://github.com/datalad-handbook/DVC-DataLad.git (git)]
>  'https://github.com/datalad-handbook/DVC-DataLad.git' configured as sibling 'github
> ↪' for Dataset(/home/me/DVCvsDL/DVC-DataLad)
> $ datalad push --to github
>   Update availability for 'github': [...] [00:00<00:00, 28.9k Steps/s]Username for
> ↪'https://github.com': <user>
>   Password for 'https://adswa@github.com': <password>
>   publish(ok): /home/me/DVCvsDL/DVC-DataLad (dataset) [refs/heads/master->
> ↪github:refs/heads/master [new branch]]
>   publish(ok): /home/me/DVCvsDL/DVC-DataLad (dataset) [refs/heads/git-annex->
> ↪github:refs/heads/git-annex [new branch]]
> ```
>
> Next, we can clone this dataset, and retrieve the files:

---

[534] In DataLad datasets, data duplication is usually avoided as GIT-ANNEX uses SYMLINKs. Only on file systems that lack support for symlinks or for removing write PERMISSIONS from files (so called "crippled file systems" such as /sdcard on Android, FAT or NTFS) git-annex needs to duplicate data.

[521] https://en.wikipedia.org/wiki/Hard_link

```
### DVC-DataLad
# outside of a dataset
$ datalad clone https://github.com/datalad-handbook/DVC-DataLad.git DVC-DataLad-2
$ cd DVC-DataLad-2
[INFO] Cloning dataset to Dataset(/home/me/DVCvsDL/DVC-DataLad-2)
[INFO] Attempting to clone from https://github.com/datalad-handbook/DVC-DataLad.git
→to /home/me/DVCvsDL/DVC-DataLad-2
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/DVCvsDL/DVC-DataLad-2)
[INFO] scanning for annexed files (this may take some time)
[INFO] Remote origin not usable by git-annex; setting annex-ignore
[INFO] https://github.com/datalad-handbook/DVC-DataLad.git/config download failed:
→Not Found
install(ok): /home/me/DVCvsDL/DVC-DataLad-2 (dataset)
```

```
### DVC-DataLad2
$ datalad get data/raw/val
[INFO] To obtain some keys we need to fetch an archive of size 15.1 MB
[INFO] datalad-archives special remote is using an extraction cache under /home/me/
→DVCvsDL/DVC-DataLad-2/.git/datalad/tmp/archives/8f2938add6. Remove it with DataLad
→'s 'clean' command to save disk space.
get(ok): data/raw/val (directory)
action summary:
  get (ok: 790)
```

The data was retrieved by re-downloading the original archive from S3 and extracting the required files.

Here's an example of pushing a dataset to a local sibling nevertheless:

Step 1: Set up the sibling

The easiest way to share data is via a local sibling[533]. This won't share only annexed data, but it instead will push everything, including the Git aspect of the dataset. First, we need to create a local sibling:

```
### DVC-DataLad
$ cd DVC-DataLad
$ datalad create-sibling --name mysibling ../datalad-sibling
[INFO] Considering to create a target dataset /home/me/DVCvsDL/DVC-DataLad at /home/me/
→DVCvsDL/datalad-sibling of localhost
[INFO] Fetching updates for Dataset(/home/me/DVCvsDL/DVC-DataLad)
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Adjusting remote git configuration
[INFO] Running post-update hooks in all created siblings
create_sibling(ok): /home/me/DVCvsDL/DVC-DataLad (dataset)
```

Step 2: Push the data

Afterwards, the dataset contents can be pushed using `datalad push`.

```
### DVC-DataLad
$ datalad push --to mysibling
[INFO] Determine push target
[INFO] Push refspecs
[INFO] Transfer data
[INFO] Update availability information
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start writing objects
[INFO] Start resolving deltas
[INFO] Finished push of Dataset(/home/me/DVCvsDL/DVC-DataLad)
publish(ok): . (dataset) [refs/heads/git-annex->mysibling:refs/heads/git-annex 363fc913..
↪a52bcf0f]
publish(ok): . (dataset) [refs/heads/master->mysibling:refs/heads/master [new branch]]
action summary:
  copy (ok: 2701)
  publish (ok: 2)
```

This pushed all of the annexed data and the Git history of the dataset.

Step 3: Retrieve the data

The data in the dataset (complete directories or individual files) can be dropped using **datalad drop**, and reobtained using **datalad get**.

```
### DVC-DataLad
$ datalad drop data/raw/val
drop(ok): data/raw/val (directory)
action summary:
  drop (ok: 790)
```

```
### DVC-DataLad
$ datalad get data/raw/val
get(ok): data/raw/val (directory)
action summary:
  get (ok: 790)
```

Data analysis

DVC is tuned towards machine learning analyses and comes with convenience commands and workflow management to build, compare, and reproduce machine learning pipelines. The tutorial therefore runs an SGD classifier and a random forrest classifier on the data and compares the two models. For this, the pre-existing preparation, training, and evaluation scripts are used on the data we have downloaded and version controlled in the previous steps. DVC has means to transform such a structured ML analysis into a workflow, reproduce this workflow on demand, and compare it across different models or parametrizations.

In this general overview, we will only rush through the analysis: In short, it consists of three steps, each associated with a script. `src/prepare.py` creates two `.csv` files with mappings of file names in `train/` and `val/` to image categories. Later, these files will be used to train and test the classifiers. `src/train.py` loads the training CSV file prepared in the previous stage, trains a classifier on the training data, and saves the classifier into the `model/` directory as `model.joblib`. The final script, `src/evaluate.py` is used to evaluate the trained classifier on the validation data and write the accuracy of the classification into the file `metrics/accuracy.json`.

There are more detailed insights and explanations of the actual analysis code in the Tutorial[522] if you're interested in finding out more.

For workflow management, DVC has the concept of a "DVC pipeline". A pipeline consists of multiple stages and is executed using a **dvc run** command. Each stage has three components: "deps", "outs", and "command". Each of the scripts in the repository will be represented by a stage in the DVC pipeline.

DataLad does not have any workflow management functions. The closest to it are **datalad run** to record any command execution or analysis, **datalad rerun** to recompute such an analysis, and **datalad containers-run** to perform and record a command execution or analysis inside of a tracked software container[536].

DVC workflow

Model 1: SGD classifier

Each model will be analyzed in a different branch of the repository. Therefore, we start by creating a new branch.

```
### DVC
$ cd ../DVC
$ git checkout -b sgd-pipeline
Switched to a new branch 'sgd-pipeline'
```

The first stage in the pipeline is data preparation (performed by the script prepare.py). The following command sets up the stage:

```
### DVC
$ dvc run -n prepare \
  -d src/prepare.py -d data/raw \
  -o data/prepared/train.csv -o data/prepared/test.csv \
  python src/prepare.py
Running stage 'prepare' with command:
        python src/prepare.py
Creating 'dvc.yaml'
Adding stage 'prepare' in 'dvc.yaml'
Generating lock file 'dvc.lock'
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add data/prepared/.gitignore dvc.yaml dvc.lock
```

The -n parameter gives the stage a name, the -d parameter passes the dependencies – the raw data – to the command, and the -o parameter defines the outputs of the command – the CSV files that prepare.py will create. python src/prepare.py is the command that will be executed in the stage.

The resulting changes can be added to Git:

```
### DVC
$ git add dvc.yaml data/prepared/.gitignore dvc.lock
```

---

[522] https://realpython.com/python-data-version-control
[536] To re-read about **datalad run** and **datalad rerun**, checkout chapter *DataLad, Run!* (page 59).

This command runs the command, and also creates two YAML[523] files, `dvc.yaml` and `dvc.lock`. They contain the pipeline description, which currently comprises of the first stage:

```
### DVC
$ cat dvc.yaml
stages:
  prepare:
    cmd: python src/prepare.py
    deps:
    - data/raw
    - src/prepare.py
    outs:
    - data/prepared/test.csv
    - data/prepared/train.csv
```

The lock file tracks the versions of all relevant files via MD5 hashes. This allows DVC to track all dependencies and outputs and detect if any of these files change.

```
### DVC
$ cat dvc.lock
prepare:
  cmd: python src/prepare.py
  deps:
  - path: data/raw
    md5: d39907b06425b95b440a692eb1af5ba4.dir
    size: 16711927
    nfiles: 2704
  - path: src/prepare.py
    md5: ef804f358e00edcfe52c865b471f8f55
    size: 1231
  outs:
  - path: data/prepared/test.csv
    md5: 536fe137c83d7119c45f5d978335425b
    size: 62023
  - path: data/prepared/train.csv
    md5: 0bad47e2449d20d62df6fd9fdbeaa32b
    size: 155128
```

The command also added the results from the stage, `train.csv` and `test.csv` into a `.gitignore` file.

The next pipeline stage is training, in which `train.py` will be used to train a classifier on the data. Initially, this classifier is an SGD classifier. The following command sets it up:

```
$ dvc run -n train \
   -d src/train.py -d data/prepared/train.csv \
   -o model/model.joblib \
   python src/train.py
Running stage 'train' with command:
       python src/train.py
/home/adina/env/handbook2/lib/python3.9/site-packages/sklearn/linear_model/_stochastic_
↪gradient.py:570: ConvergenceWarning: Maximum number of iteration reached before␣
↪convergence. Consider increasing max_iter to improve the fit.
  warnings.warn("Maximum number of iteration reached before "
Adding stage 'train' in 'dvc.yaml'
```

(continues on next page)

---

[523] https://en.wikipedia.org/wiki/YAML

```
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.yaml model/.gitignore dvc.lock
```

Afterwards, `train.py` has been executed, and the pipelines have been updated with a second stage. The resulting changes can be added to Git:

```
### DVC
$ git add dvc.yaml model/.gitignore dvc.lock
```

Finally, we create the last stage, model evaluation. The following command sets it up:

```
$ dvc run -n evaluate \
        -d src/evaluate.py -d model/model.joblib \
        -M metrics/accuracy.json \
        python src/evaluate.py
Running stage 'evaluate' with command:
        python src/evaluate.py
Adding stage 'evaluate' in 'dvc.yaml'
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.yaml dvc.lock


### DVC
$ git add dvc.yaml dvc.lock
```

Instead of "outs", this final stage uses the `-M` flag to denote a "metric". This type of flag can be used if floating-point or integer values that summarize model performance (e.g. accuracies, receiver operating characteristics, or area under the curve values) are saved in hierarchical files (JSON, YAML). DVC can then read from these files to display model performances and comparisons:

```
### DVC
$ dvc metrics show
        metrics/accuracy.json:
                accuracy: 0.8022813688212928
```

The complete pipeline now consists of preparation, training, and evaluation. It now needs to be committed, tagged, and pushed:

```
### DVC
$ git add --all
$ git commit -m "Add SGD pipeline"
$ dvc commit
$ git push --set-upstream origin sgd-pipeline
$ git tag -a sgd-pipeline -m "Trained SGD as DVC pipeline."
$ git push origin --tags
$ dvc push
[sgd-pipeline 0268557] Add SGD pipeline
 5 files changed, 71 insertions(+)
```

```
 create mode 100644 dvc.lock
 create mode 100644 dvc.yaml
 create mode 100644 metrics/accuracy.json
error: src refspec sgd-pipeline matches more than one
error: failed to push some refs to '/home/me/pushes/data-version-control'
fatal: tag 'sgd-pipeline' already exists
To /home/me/pushes/data-version-control
 * [new tag]           random-forest -> random-forest
 * [new tag]           sgd-pipeline -> sgd-pipeline
3 files pushed
```

Model 2: random forrest classifier

In order to explore a second model, a random forrest classifier, we start with a new branch.

```
### DVC
$ git checkout -b random_forrest
Switched to a new branch 'random_forrest'
```

To switch from SGD to a random forrest classifier, a few lines of code within `train.py` need to be changed. The following here doc[524] changes the script accordingly (changes are highlighted):

```
### DVC
$ cat << EOT >| src/train.py
from joblib import dump
from pathlib import Path

import numpy as np
import pandas as pd
from skimage.io import imread_collection
from skimage.transform import resize
from sklearn.ensemble import RandomForestClassifier

def load_images(data_frame, column_name):
    filelist = data_frame[column_name].to_list()
    image_list = imread_collection(filelist)
    return image_list

def load_labels(data_frame, column_name):
    label_list = data_frame[column_name].to_list()
    return label_list

def preprocess(image):
    resized = resize(image, (100, 100, 3))
    reshaped = resized.reshape((1, 30000))
    return reshaped

def load_data(data_path):
    df = pd.read_csv(data_path)
    labels = load_labels(data_frame=df, column_name="label")
    raw_images = load_images(data_frame=df, column_name="filename")
    processed_images = [preprocess(image) for image in raw_images]
    data = np.concatenate(processed_images, axis=0)
    return data, labels
```

---

[524] https://en.wikipedia.org/wiki/Here_document

```
def main(repo_path):
    train_csv_path = repo_path / "data/prepared/train.csv"
    train_data, labels = load_data(train_csv_path)
    rf = RandomForestClassifier()
    trained_model = rf.fit(train_data, labels)
    dump(trained_model, repo_path / "model/model.joblib")


if __name__ == "__main__":
    repo_path = Path(__file__).parent.parent
    main(repo_path)
EOT
```

Afterwards, since `train.py` is changed, **dvc status** will realize that one dependency of the pipeline stage "train" has changed:

```
### DVC
$ dvc status
train:
        changed deps:
                modified:           src/train.py
```

Since the code change (stage 2) will likely affect the metric (stage 3), its best to reproduce the whole chain. You can reproduce a complete DVC pipeline file with the **dvc repro <stagename>** command:

```
### DVC
$ dvc repro evaluate
'data/raw/val.dvc' didn't change, skipping
'data/raw/train.dvc' didn't change, skipping
Stage 'prepare' didn't change, skipping
Running stage 'train' with command:
        python src/train.py
Updating lock file 'dvc.lock'

Running stage 'evaluate' with command:
        python src/evaluate.py
Updating lock file 'dvc.lock'

To track the changes with git, run:

        git add dvc.lock
Use `dvc push` to send your updates to remote storage.
```

DVC checks the dependencies of the pipeline and re-executes commands that need to be executed again. Compared to the branch `sgd_pipeline`, the workspace in the current `random_forrest` branch contains a changed script (`src/train.py`), a changed trained classifier (`model/model.joblib`), and a changed metric (`metric/accuracy.json`). All these changes need to be committed, tagged, and pushed now.

```
### DVC
$ git add --all
$ git commit -m "Train Random Forrest classifier"
$ dvc commit
$ git push --set-upstream origin random-forest
```

```
$ git tag -a random-forest -m "Random Forest classifier with 80.99% accuracy."
$ git push origin --tags
$ dvc push
[random_forrest 6890032] Train Random Forrest classifier
 3 files changed, 11 insertions(+), 17 deletions(-)
Everything up-to-date
fatal: tag 'random-forest' already exists
Everything up-to-date
1 file pushed
```

At this point, you can compare metrics across multiple tags:

```
### DVC
$ dvc metrics show -T
workspace:
        metrics/accuracy.json:
                accuracy: 0.8048162230671736
random-forest:
        metrics/accuracy.json:
                accuracy: 0.8187579214195184
sgd-pipeline:
        metrics/accuracy.json:
                accuracy: 0.7427122940430925
```

Done!

DataLad workflow

For a direct comparison to DVC, we'll try to mimic the DVC workflow as closely as it is possible with DataLad.

Model 1: SGD classifier

```
### DVC-DataLad
$ cd ../DVC-DataLad
```

As there is no workflow manager in DataLad[535], each script execution needs to be done separately. To record the execution, get all relevant inputs, and recompute outputs at later points, we can set up a **datalad run** call[536]. Later on, we can rerun a range of **datalad run** calls at once to recompute the relevant aspects of the analysis. To harmonize execution and to assist with reproducibility of the results, we generally recommend to create a container (Docker or Singularity), add it to the repository as well, and use **datalad containers-run** call[537] and have that reran, but we'll stay basic here.

Let's start with data preparation. Instead of creating a pipeline stage and giving it a name, we attach a meaningful commit message.

```
### DVC-DataLad
$ datalad run --message "Prepare the train and testing data" \
   --input "data/raw/*" \
```

---

[535] yet.

[537] To re-read about joining code, execution, data, results and software environment in a re-executable record with **datalad container-run**, checkout section *Computational reproducibility with software containers* (page 166).

```
     --output "data/prepared/*" \
     python code/prepare.py
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): data/prepared/test.csv (file)
add(ok): data/prepared/train.csv (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 1, ok: 3926)
  save (ok: 1)
```

The results of this computation are automatically saved and associated with their inputs and command execution. This information isn't stored in a separate file, but in the Git history, and saved with the commit message we have attached to the **run** command.

To stay close to the DVC tutorial, we will also work with tags to identify analysis versions, but DataLad could also use a range of other identifiers, for example commit hashes, to identify this computation. As we at this point have set up our data and are ready for the analysis, we will name the first tag "ready-for-analysis". This can be done with **git tag**, but also with **datalad save**.

```
### DVC-DataLad
$ datalad save --version-tag ready-for-analysis
save(ok): . (dataset)
```

Let's continue with training by running code/train.py on the prepared data.

```
### DVC-DataLad
$ datalad run --message "Train an SGD classifier" \
     --input "data/prepared/*" \
     --output "model/model.joblib" \
     python code/train.py
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
/home/adina/env/handbook2/lib/python3.9/site-packages/sklearn/linear_model/_stochastic_
↪gradient.py:574: ConvergenceWarning: Maximum number of iteration reached before␣
↪convergence. Consider increasing max_iter to improve the fit.
  warnings.warn("Maximum number of iteration reached before "
[INFO] == Command exit (modification check follows) =====
add(ok): model/model.joblib (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 2)
  save (ok: 1)
```

As before, the results of this computations are saved, an the Git history connects computation, results, and inputs.

As a last step, we evaluate the first model:

```
### DVC-DataLad
$ datalad run --message "Evaluate SGD classifier model" \
```

```
  --input "model/model.joblib" \
  --output "metrics/accuracy.json" \
  python code/evaluate.py
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): code/__pycache__/train.cpython-39.pyc (file)
add(ok): metrics/accuracy.json (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 1)
  save (ok: 1)
```

At this point, the first accuracy metric is saved in `metrics/accuracy.json`. Let's add a tag to declare that it belongs to the SGD classifier.

```
### DVC-DataLad
$ datalad save --version-tag SGD
save(ok): . (dataset)
```

Let's now change the training script to use a random forrest classifier as before:

```
### DVC-DataLad
$ cat << EOT >| code/train.py
from joblib import dump
from pathlib import Path

import numpy as np
import pandas as pd
from skimage.io import imread_collection
from skimage.transform import resize
from sklearn.ensemble import RandomForestClassifier

def load_images(data_frame, column_name):
    filelist = data_frame[column_name].to_list()
    image_list = imread_collection(filelist)
    return image_list

def load_labels(data_frame, column_name):
    label_list = data_frame[column_name].to_list()
    return label_list

def preprocess(image):
    resized = resize(image, (100, 100, 3))
    reshaped = resized.reshape((1, 30000))
    return reshaped

def load_data(data_path):
    df = pd.read_csv(data_path)
    labels = load_labels(data_frame=df, column_name="label")
    raw_images = load_images(data_frame=df, column_name="filename")
    processed_images = [preprocess(image) for image in raw_images]
    data = np.concatenate(processed_images, axis=0)
    return data, labels
```

```
def main(repo_path):
    train_csv_path = repo_path / "data/prepared/train.csv"
    train_data, labels = load_data(train_csv_path)
    rf = RandomForestClassifier()
    trained_model = rf.fit(train_data, labels)
    dump(trained_model, repo_path / "model/model.joblib")

if __name__ == "__main__":
    repo_path = Path(__file__).parent.parent
    main(repo_path)
EOT
```

We need to save this change:

```
$ datalad save -m "Switch to random forrest classification" code/train.py
add(ok): code/train.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Afterwards, we can rerun all run records between the tags `ready-for-analysis` and `SGD` using **datalad rerun**. We could automatically compute this on a different branch if we wanted to by using the branch option:

```
$ datalad rerun --branch="randomforrest" -m "Recompute classification with random forrest␣
↪classifier" ready-for-analysis..SGD
[INFO] checkout commit 2f50499;
[INFO] run commit 152599e; (Train an SGD clas...)
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): model/model.joblib (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): model/model.joblib (file)
save(ok): . (dataset)
[INFO] run commit fbc0ddf; (Evaluate SGD clas...)
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): code/__pycache__/train.cpython-39.pyc (file)
add(ok): metrics/accuracy.json (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  get (notneeded: 3)
  save (ok: 2)
  unlock (notneeded: 2, ok: 1)
```

Done! The difference in accuracies between models could now for example be compared with a `git diff`:

```
$ git diff SGD -- metrics/accuracy.json
diff --git a/metrics/accuracy.json b/metrics/accuracy.json
index f847bc7c..38044953 100644
--- a/metrics/accuracy.json
```

```
+++ b/metrics/accuracy.json
@@ -1 +1 @@
-{"accuracy": 0.752851711026616}
\ No newline at end of file
+{"accuracy": 0.8136882129277566}
\ No newline at end of file
```

Even though there is no one-to-one correspondence between a DVC and a DataLad workflow, a DVC workflow can also be implemented with DataLad.

Summary

DataLad and DVC aim to solve the same problems: Version control data, sharing data, and enabling reproducible analyses. DataLad provides generic solutions to these issues, while DVC is tuned for machine-learning pipelines. Despite their similar purpose, the looks, feels and functions of both tools are different, and its a personal decision which one you feel more comfortable with. Using DVC requires solid knowledge of Git, because DVC workflows heavily rely on effective Git practices, such as branching, tags, and `.gitignore` files. But despite the reliance on Git, DVC barely integrates with Git – changes done to files in DVC can not be detected by Git and vice versa, DVC and Git aspects of a repository have to be handled in parallel by the user, and DVC and Git have distinct command functions and concepts that nevertheless share the same name. Thus, DVC users need to master Git *and* DVC workflows and intertwine them correctly. In return, DVC provides users with workflow management and reporting tuned to machine learning analyses. It also provides a somewhat more lightweight and uniform across operating and file systems approach to "data version control" than git-annex used by DataLad.

DATALAD INTERNALS
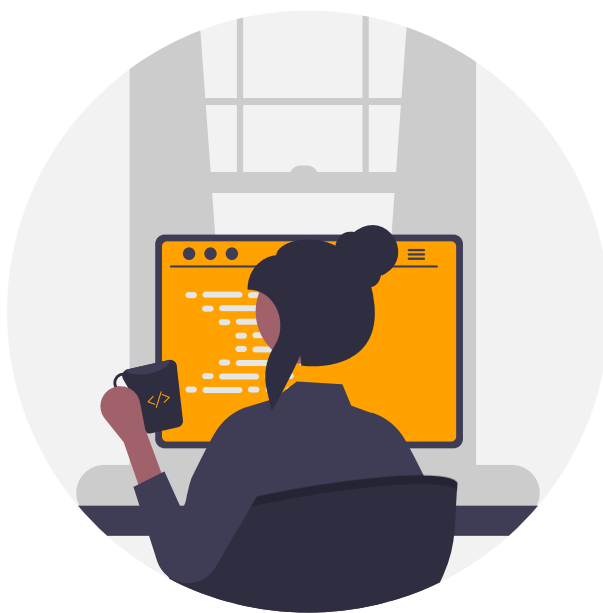
20.1 DataLad's internal design
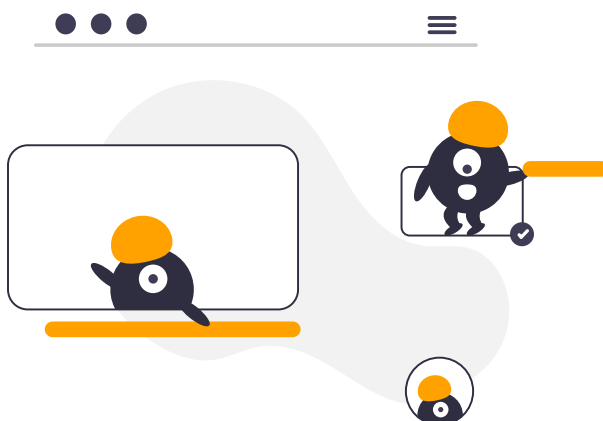
The handbook should have provided plenty of insights into common usage patterns for DataLad. When approaching the software not from a user perspective, but from a developer angle, you can find further information on the design principles and internal code structure in DataLad's Design Documents. These documents are part of the developer docs at docs.datalad.org[538] and

---

[538] http://docs.datalad.org

constitute an ongoing effort to document and harmonize the development rules and processes. Head over to docs.datalad.org/design/index.html[539] to find out more.

20.2 Contributing to DataLad

DataLad is free and open source software. Everyone can contribute in various forms – feature requests, questions, artwork, tutorials, code patches, bug reports, ... even follows, likes, or retweets on Twitter[540], or discussions in our matrix chatroom[541]. We would be delighted to hear from you in any form.

The following resources could be helpful:

For the Handbook

- Take a look at the section *Contributing* (page 498) for more information.

For DataLad

- **Use it!** Although it may sound nothing like a contribution, *using* DataLad is a fundamental contribution anyone can make. You can find further tutorials, materials, videos, and other resources in this handbook, and in a dedicated Tutorials repository[542]. And if you like it, you can also tell your friends, system administrators, and colleagues about it, or convince your local IT department to install it on shared compute infrastructure.

- **Get in touch**: We strive to improve the clarity of DataLad and its documentation. If you tried to implement DataLad in a specific way and the existing documentation didn't make sense, or wasn't clear enough or even confusing, please help us fix it. Let us know that the instructions could have been clearer, or that it didn't cover your use case, or led you along the wrong path. And if you have suggestions for improvements, let's incorporate them! Come chat with us about what you do on Matrix[543] (a free, decentralized, and secure

---

[539] http://docs.datalad.org/en/stable/design/index.html
[540] https://twitter.com/datalad
[541] https://app.element.io/#/room/%23datalad:matrix.org
[542] https://github.com/datalad/tutorials
[543] https://app.element.io/#/room/%23datalad:matrix.org

communication network), tag `datalad` in an issue on Neurostars[544], or get in touch via GITHUB.

- **Show your support**: If you like DataLad you can show your support in various meaningful ways. You can "star"[545] the project on GitHub. You can subscribe, like, or follow DataLad on social media: There is a Twitter Account[546] on which we regularly post updates, and a YouTube channel[547] on which we post tutorials and talks. And if you write academic papers or blog posts, you can cite the paper about DataLad[548] if DataLad assisted in your work.

- **Contribute on GitHub**: A most valuable contribution is your time. We are interested and grateful for opinions, bug reports, feature requests, patches, larger code contributions, or simply a notice what you use DataLad for. Find the relevant repository, be that github.com/datalad/datalad[549] (the main repository), github.com/datalad-datasets[550] (many open datasets), or any DATALAD EXTENSION, and open issues or pull requests. DataLad's CONTRIBUTING[551] file has tons of technical and social information to get you started with code contributions. But don't be intimidated by the wealth of information you will find in there. We'll be happy to help you at any stage. Also, you can take a look at technical docs (docs.datalad.org[552]) and in particular the *Design documents* (page 383) that shed light on the internal design principles of the software.

- **Write an extension!** If you have unique use cases, you can write your own DATA-LAD EXTENSION for it, that can provide any number of additional DataLad commands that are automatically included in DataLad's command line and Python API. Our extension template[553] is the best starting point. It contains an example command implementation, and will have test setup and packaging configurations in place already. If you want to, you can register your extension against DataLad's extension registry at github.com/datalad/datalad-extensions[554] – if your project is included, we can continuously check whether current versions of DataLad work with your extension.

- **Contribute to related projects** As open source software, we proudly stand on the shoulders of giants. The DataLad project wouldn't be possible without many other open source packages and projects. Helping them helps us, and you could do so in any of the ways described above, including documentation, tutorials, patches, support – if you have a passion for Haskell[555] or C[556] you could even head over to git-annex[557] or Git[558] themselves.

Thank you for your interest and support!

---

[544] https://neurostars.org/
[545] https://github.com/datalad/datalad/stargazers
[546] https://twitter.com/datalad
[547] https://youtube.com/datalad
[548] https://joss.theoj.org/papers/10.21105/joss.03262
[549] https://github.com/datalad/datalad
[550] https://github.com/datalad-datasets
[551] https://github.com/datalad/datalad/blob/master/CONTRIBUTING.md
[552] http://docs.datalad.org/
[553] https://github.com/datalad/datalad-extension-template
[554] https://github.com/datalad/datalad-extensions
[555] https://www.haskell.org/
[556] https://en.wikipedia.org/wiki/C_(programming_language)
[557] http://source.git-annex.branchable.com/?p=source.git;a=summary
[558] https://github.com/git/git

Part IV

Use cases

In this part of the book you will find concrete examples of DataLad applications for general inspiration. You can get an overview of what is possible by browsing through them, and step-by-step solutions for a range of problems in every single one. Provided you have read the previous *Basics* (page 33) sections, the usecases' code examples are sufficient (though sparser than in Basics) to recreate or apply the solutions they demonstrate.

A TYPICAL COLLABORATIVE DATA MANAGEMENT WORKFLOW

This use case sketches the basics of a common, collaborative data management workflow for an analysis:

1. A 3rd party dataset is obtained to serve as input for an analysis.

2. Data processing is collaboratively performed by two colleagues.

3. Upon completion, the results are published alongside the original data for further consumption.

The data types and methods mentioned in this usecase belong to the scientific field of neuroimaging, but the basic workflow is domain-agnostic.

21.1 The Challenge

Bob is a new PhD student and about to work on his first analysis. He wants to use an open dataset as the input for his analysis, so he asks a friend who has worked with the same dataset for the data and gets it on a hard drive. Later, he's stuck with his analysis. Luckily, Alice, a senior grad student in the same lab, offers to help him. He sends his script to her via email and hopes she finds the solution to his problem. She responds a week later with the fixed script, but in the meantime Bob already performed some miscellaneous changes to his script as well. Identifying and integrating her fix into his slightly changed script takes him half a day. When he finally finishes his analysis, he wants to publish code and data online, but can not find a way to share his data together with his code.

21.2 The DataLad Approach

Bob creates his analysis project as a DataLad dataset. Complying with the *YODA principles* (page 136), he creates his scripts in a dedicated code/ directory, and clones the open dataset as a standalone DataLad subdataset within a dedicated subdirectory. To collaborate with his senior grad student Alice, he shares the dataset on the lab's SSH server, and they can collaborate on the version controlled dataset almost in real time with no need for Bob to spend much time integrating the fix that Alice provides him with. Afterwards, Bob can execute his scripts in a way that captures all provenance for this results with a `datalad run` command. Bob can share his whole project after completion by creating a sibling on a webserver, and pushing all of his dataset, including the input data, to this sibling, for everyone to access and recompute.

21.3 Step-by-Step

Bob creates a DataLad dataset for his analysis project to live in. Because he knows about the YODA principles, he configures the dataset to be a YODA dataset right at the time of creation:

```
$ datalad create -c yoda --description "my 1st phd project on work computer" myanalysis
[INFO] Creating a new annex repo at /home/me/usecases/collab/myanalysis
[INFO] scanning for unlocked files (this may take some time)
[INFO] Running procedure cfg_yoda
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
create(ok): /home/me/usecases/collab/myanalysis (dataset)
```

After creation, there already is a code/ directory, and all of its inputs are version-controlled by GIT instead of GIT-ANNEX thanks to the yoda procedure:

```
$ cd myanalysis
$ tree
.
├── CHANGELOG.md
├── code
│   └── README.md
└── README.md

1 directory, 3 files
```

Bob knows that a DataLad dataset can contain other datasets. He also knows that as any content of a dataset is tracked and its precise state is recorded, this is a powerful method to specify and later resolve data dependencies, and that including the dataset as a standalone data component will it also make it easier to keep his analysis organized and share it later. The dataset that Bob wants to work with is structural brain imaging data from the studyforrest project[559], a public data resource that the original authors share as a DataLad dataset through GITHUB. This means that Bob can simply clone the relevant dataset from this service and into his own dataset. To do that, he clones it as a subdataset into a directory he calls src/ as he wants to make it obvious which parts of his analysis steps and code require 3rd party data:

```
$ datalad clone -d . https://github.com/psychoinformatics-de/studyforrest-data-structural.
↪git src/forrest_structural
[INFO] Cloning dataset to Dataset(/home/me/usecases/collab/myanalysis/src/forrest_
↪structural)
[INFO] Attempting to clone from https://github.com/psychoinformatics-de/studyforrest-data-
↪structural.git to /home/me/usecases/collab/myanalysis/src/forrest_structural
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/usecases/collab/myanalysis/src/
↪forrest_structural)
[INFO] scanning for unlocked files (this may take some time)
[INFO] Remote origin not usable by git-annex; setting annex-ignore
install(ok): src/forrest_structural (dataset)
add(ok): src/forrest_structural (file)
```

(continues on next page)

---

[559] http://studyforrest.org/

```
add(ok): .gitmodules (file)
save(ok): . (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  install (ok: 1)
  save (ok: 2)
```

Now that he executed this command, Bob has access to the entire dataset content, and the precise version of the dataset got linked to his top-level dataset myanalysis. However, no data was actually downloaded (yet). Bob very much appreciates that DataLad datasets primarily contain information on a dataset's content and where to obtain it: Cloning above was done rather quickly, and will still be relatively lean even for a dataset that contains several hundred GBs of data. He knows that his script can obtain the relevant data he needs on demand if he wraps it into a **datalad run** command and therefore does not need to care about getting the data yet. Instead, he focuses to write his script code/run_analysis.sh. To save this progress, he runs frequent **datalad save** commands:

```
$ datalad save -m "First steps: start analysis script" code/run_analysis.py
add(ok): code/run_analysis.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Once Bob's analysis is finished, he can wrap it into **datalad run**. To ease execution, he first makes his script executable by adding a SHEBANG that specifies Python as an interpreter at the start of his script, and giving it executable PERMISSIONS:

```
$ chmod +x code/run_analysis.py
$ datalad save -m "make script executable"
add(ok): code/run_analysis.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Importantly, prior to a **datalad run**, he specifies the necessary inputs such that DataLad can take care of the data retrieval for him:

```
$ datalad run -m "run first part of analysis workflow" \
  --input "src/forrest_structural" \
  --output results.txt \
  "code/run_analysis.py"
[INFO] Making sure inputs are available (this may take some time)
get(ok): src/forrest_structural/sub-01/anat/sub-01_T1w.nii.gz (file) [from mddatasrc...]
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
```

This will take care of retrieving the data, running Bobs script, and saving all outputs.

Some time later, Bob needs help with his analysis. He turns to his senior grad student Alice for help. Alice and Bob both work on the same computing server. Bob has told Alice in which directory he keeps his analysis dataset, and the directory is configured to have PERMISSIONS

that allow for read-access for all lab-members, so Alice can obtain Bob's work directly from his home directory:

```
$ datalad clone /myanalysis bobs_analysis
[INFO] Cloning dataset to Dataset(/home/me/usecases/collab/bobs_analysis)
[INFO] Attempting to clone from myanalysis to /home/me/usecases/collab/bobs_analysis
[INFO] Completed clone attempts for Dataset(/home/me/usecases/collab/bobs_analysis)
[INFO] scanning for unlocked files (this may take some time)
install(ok): /home/me/usecases/collab/bobs_analysis (dataset)
```

```
$ cd bobs_analysis
# ... make contributions, and save them
$ [...]
$ datalad save -m "you're welcome, bob"
add(ok): code/run_analysis.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Alice can get the studyforrest data Bob used as an input as well as the result file, but she can also rerun his analysis by using **datalad rerun**. She goes ahead and fixes Bobs script, and saves the changes. To integrate her changes into his dataset, Bob registers Alice's dataset as a sibling:

```
#in Bobs home directory
$ datalad siblings add -s alice --url '/bobs_analysis'
.: alice(+) [../bobs_analysis (git)]
```

Afterwards, he can get her changes with a **datalad update --merge** command:

```
$ datalad update -s alice --merge
[INFO] Fetching updates for Dataset(/home/me/usecases/collab/myanalysis)
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
merge(ok): . (dataset) [Merged alice/master]
update.annex_merge(ok): . (dataset) [Merged annex branch]
update(ok): . (dataset)
action summary:
  merge (ok: 1)
  update (ok: 1)
  update.annex_merge (ok: 1)
```

Finally, when Bob is ready to share his results with the world or a remote collaborator, he makes his dataset available by uploading them to a webserver via SSH. Bob does so by creating a sibling for the dataset on the server, to which the dataset can be published and later also updated.

```
# this generated sibling for the dataset and all subdatasets
$ datalad create-sibling --recursive -s public "$SERVER_URL"
```

Once the remote sibling is created and registered under the name "public", Bob can publish his version to it.

```
$ datalad push -r --to public .
```

This workflow allowed Bob to obtain data, collaborate with Alice, and publish or share his dataset with others easily – he cannot wait for his next project, given that this workflow made his life so simple.

BASIC PROVENANCE TRACKING

This use case demonstrates how the provenance of downloaded and generated files can be captured with DataLad by

1. downloading a data file from an arbitrary URL from the web

2. perform changes to this data file and

3. capture provenance for all of this

> ⚠️ **How to become a Git pro**
>
> This section uses advanced Git commands and concepts on the side that are not covered in the book. If you want to learn more about the Git commands shown here, the ProGit book[560] is an excellent resource.
>
> ---
> [560] https://git-scm.com/book/en/v2

22.1 The Challenge

Rob needs to turn in an art project at the end of the high school year. He wants to make it as easy as possible and decides to just make a photomontage of some pictures from the internet. When he submits the project, he does not remember where he got the input data from, nor the exact steps to create his project, even though he tried to take notes.

22.2 The DataLad Approach

Rob starts his art project as a DataLad dataset. When downloading the images he wants to use for his project, he tracks where they come from. And when he changes or creates output, he tracks how, when and why and this was done using standard DataLad commands. This will make it easy for him to find out or remember what he has done in his project, and how it has been done, a long time after he finished the project, without any note taking.

22.3 Step-by-Step

Rob starts by creating a dataset, because everything in a dataset can be version controlled and tracked:

```
$ datalad create artproject && cd artproject
[INFO] Creating a new annex repo at /home/me/usecases/provenance/artproject
[INFO] scanning for unlocked files (this may take some time)
create(ok): /home/me/usecases/provenance/artproject (dataset)
```

For his art project, Rob decides to download a mosaic image composed of flowers from Wikimedia. As a first step, he extracts some of the flowers into individual files to reuse them later. He uses the **datalad download-url** command to get the resource straight from the web, but also capture all provenance automatically, and save the resource in his dataset together with a useful commit message:

```
$ mkdir sources
$ datalad download-url -m "Added flower mosaic from wikimedia" \
  https://upload.wikimedia.org/wikipedia/commons/a/a5/Flower_poster_2.jpg \
  --path sources/flowers.jpg
[INFO] Downloading 'https://upload.wikimedia.org/wikipedia/commons/a/a5/Flower_poster_2.
↪jpg' into '/home/me/usecases/provenance/artproject/sources/flowers.jpg'
download_url(ok): /home/me/usecases/provenance/artproject/sources/flowers.jpg (file)
add(ok): sources/flowers.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
```

If he later wants to find out where he obtained this file from, a **git annex whereis**[562] command will tell him:

```
$ git annex whereis sources/flowers.jpg
whereis sources/flowers.jpg (2 copies)
        00000000-0000-0000-0000-000000000001 -- web
        6a5e5a27-06b0-4987-b028-c9fcfa0d13c4 -- me@muninn:~/usecases/provenance/
↪artproject [here]

  web: https://upload.wikimedia.org/wikipedia/commons/a/a5/Flower_poster_2.jpg
ok
```

To extract some image parts for the first step of his project, he uses the extract tool from ImageMagick[561] to extract the St. Bernard's Lily from the upper left corner, and the pimpernel from the upper right corner. The commands will take the Wikimedia poster as an input and produce output files from it. To capture provenance on this action, Rob wraps it into **datalad run**[563] commands.

```
$ datalad run -m "extract st-bernard lily" \
 --input "sources/flowers.jpg" \
 --output "st-bernard.jpg" \
 "convert -extract 1522x1522+0+0 sources/flowers.jpg st-bernard.jpg"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): st-bernard.jpg (file)
save(ok): . (dataset)
```

(continues on next page)

---

[562] If you want to learn more about **git annex whereis**, re-read section *Where's Waldo?* (page 99).
[561] https://imagemagick.org/index.php
[563] If you want to learn more about **datalad run**, read on from section *Keeping track* (page 59).

```
action summary:
  add (ok: 1)
  get (notneeded: 1)
  save (ok: 1)


$ datalad run -m "extract pimpernel" \
  --input "sources/flowers.jpg" \
  --output "pimpernel.jpg" \
  "convert -extract 1522x1522+1470+1470 sources/flowers.jpg pimpernel.jpg"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): pimpernel.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 1)
  save (ok: 1)
```

He continues to process the images, capturing all provenance with DataLad. Later, he can always find out which commands produced or changed which file. This information is easily accessible within the history of his dataset, both with Git and DataLad commands such as **git log** or **datalad diff**.

```
$ git log --oneline HEAD~3..HEAD
ee60856 [DATALAD RUNCMD] extract pimpernel
604e0ec [DATALAD RUNCMD] extract st-bernard lily
8abceb1 Added flower mosaic from wikimedia


$ datalad diff -f HEAD~3
    added: pimpernel.jpg (file)
    added: sources/flowers.jpg (file)
    added: st-bernard.jpg (file)
```

Based on this information, he can always reconstruct how an when any data file came to be – across the entire life-time of a project.

He decides that one image manipulation for his art project will be to displace pixels of an image by a random amount to blur the image:

```
$ datalad run -m "blur image" \
   --input "st-bernard.jpg" \
   --output "st-bernard-displaced.jpg" \
   "convert -spread 10 st-bernard.jpg st-bernard-displaced.jpg"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): st-bernard-displaced.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 1)
  save (ok: 1)
```

Because he is not completely satisfied with the first random pixel displacement, he decides to

retry the operation. Because everything was wrapped in **datalad run**, he can rerun the command. Rerunning the command will produce a commit, because the displacement is random and the output file changes slightly from its previous version.

```
$ git log -1 --oneline HEAD
548ba71 [DATALAD RUNCMD] blur image
```

```
$ datalad rerun 548ba71b3f319d7f88b85ba899081245c567b31c
[INFO] run commit 548ba71; (blur image)
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): st-bernard-displaced.jpg (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): st-bernard-displaced.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 1)
  save (ok: 1)
  unlock (ok: 1)
```

This blur also does not yet fulfill Robs expectations, so he decides to discard the change, using standard Git tools[564].

```
$ git reset --hard HEAD~1
HEAD is now at 548ba71 [DATALAD RUNCMD] blur image
```

He knows that within a DataLad dataset, he can also rerun *a range* of commands with the --since flag, and even specify alternative starting points for rerunning them with the --onto flag. Every command from commits reachable from the specified checksum until --since (but not including --since) will be re-executed. For example, datalad rerun --since=HEAD~5 will re-execute any commands in the last five commits. --onto indicates where to start rerunning the commands from. The default is HEAD, but anything other than HEAD will be checked out prior to execution, such that re-execution happens in a detached HEAD state, or checked out out on the new branch specified by the --branch flag. If --since is an empty string, it is set to rerun every command from the first commit that contains a recorded command. If --onto is an empty string, re-execution is performed on top to the parent of the first run commit in the revision list specified with --since. When both arguments are set to empty strings, it therefore means "rerun all commands with HEAD at the parent of the first commit a command". In other words, Rob can "replay" all the history for his artproject in a single command. Using the --branch option of **datalad rerun**, he does it on a new branch he names replay:

```
$ datalad rerun --since= --onto= --branch=replay
[INFO] checkout commit 8abceb1;
[INFO] run commit 604e0ec; (extract st-bernar...)
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): st-bernard.jpg (file)
save(ok): . (dataset)
[INFO] run commit ee60856; (extract pimpernel)
[INFO] Making sure inputs are available (this may take some time)
```

(continues on next page)

---

[564] Find out more about working with the history of a dataset with Git in section *Miscellaneous file system operations* (page 224)

```
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): pimpernel.jpg (file)
save(ok): . (dataset)
[INFO] run commit 548ba71; (blur image)
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): st-bernard-displaced.jpg (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  get (notneeded: 3)
  save (ok: 3)
```

Now he is on a new branch of his project, which contains "replayed" history.

```
$ git log --oneline --graph master replay
* 5a31ed6 [DATALAD RUNCMD] blur image
* b67c608 [DATALAD RUNCMD] extract pimpernel
* 5a861fa [DATALAD RUNCMD] extract st-bernard lily
| * 548ba71 [DATALAD RUNCMD] blur image
| * ee60856 [DATALAD RUNCMD] extract pimpernel
| * 604e0ec [DATALAD RUNCMD] extract st-bernard lily
|/
* 8abceb1 Added flower mosaic from wikimedia
* 62829d6 [DATALAD] new dataset
```

He can even compare the two branches:

```
$ datalad diff -t master -f replay
 modified: st-bernard-displaced.jpg (file)
```

He can see that the blurring, which involved a random element, produced different results. Because his dataset contains two branches, he can compare the two branches using normal Git operations. The next command, for example, marks which commits are "patch-equivalent" between the branches. Notice that all commits are marked as equivalent (=) except the 'random spread' ones.

```
$ git log --oneline --left-right --cherry-mark master...replay
> 5a31ed6 [DATALAD RUNCMD] blur image
= b67c608 [DATALAD RUNCMD] extract pimpernel
= 5a861fa [DATALAD RUNCMD] extract st-bernard lily
< 548ba71 [DATALAD RUNCMD] blur image
= ee60856 [DATALAD RUNCMD] extract pimpernel
= 604e0ec [DATALAD RUNCMD] extract st-bernard lily
```

Rob can continue processing images, and will turn in a successful art project. Long after he finishes high school, he finds his dataset on his old computer again and remembers this small project fondly.

WRITING A REPRODUCIBLE PAPER

This use case demonstrates how to use nested DataLad datasets to create a fully reproducible paper by linking

1. (different) DataLad dataset sources with

2. the code needed to compute results and

3. LaTeX files to compile the resulting paper.

The different components each exist in individual DataLad datasets and are aggregated into a single DataLad superdataset complying to the YODA principles for data analysis projects[573]. The resulting superdataset can be publicly shared, data can be obtained effortlessly on demand by anyone that has the superdataset, and results and paper can be generated and recomputed everywhere on demand.

A template to start your own reproducible paper with the same set up can be found on GitHub[565].

23.1 The Challenge

Over the past year, Steve worked on the implementation of an algorithm as a software package. For testing purposes, he used one of his own data collections, and later also included a publicly shared data collection. After completion, he continued to work on validation analyses to prove the functionality and usefulness of his software. Next to a directory in which he developed his code, and directories with data he tested his code on, he now also has other directories with different data sources used for validation analyses. "This can not take too long!" Steve thinks optimistically when he finally sits down to write up a paper.

His scripts run his algorithm on the different data collections, create derivatives of his raw data, pretty figures, and impressive tables. Just after he hand-copies and checks the last decimal of the final result in the very last table of his manuscript, he realizes that the script specified the wrong parameter values, and all of the results need to be recomputed - and obviously updated in his manuscript. When writing the discussion, he finds a paper that reports an error in the publicly shared data collection he uses. After many more days of updating tables and fixing data columns by hand, he finally submits the paper. Trying to stand with his values of open and reproducible science, he struggles to bundle all scripts, algorithm code, and data he used in a shareable form, and frankly, with all the extra time this manuscript took him so far, he lacks motivation and time. In the end, he writes a three page long README file in his GitHub

---

[573] You can read up on the YODA principles again in section *YODA: Best practices for data analyses in a dataset* (page 136)

[565] https://github.com/datalad-handbook/repro-paper-sketch/

code repository, includes his email for data requests, and secretly hopes that no-one will want to recompute his results, because by now even he himself forgot which script ran on which dataset and what data was fixed in which way, or whether he was careful enough to copy all of the results correctly. In the review process, reviewer 2 demands that the figures his software produces need to get a new color scheme, which requires updates in his software package, and more recomputations.

23.2 The DataLad Approach

Steve sets up a DataLad dataset and calls it `algorithm-paper`. In this dataset, he creates several subdirectories to collate everything that is relevant for the manuscript: Data, code, a manuscript backbone without results. `code/` contains a Python script that he uses for validation analyses, and prior to computing results, the script attempts to download the data should the files need to be obtained using DataLad's Python API. `data/` contains a separate DataLad subdataset for every dataset he uses. An `algorithm/` directory is a DataLad dataset containing a clone of his software repository, and within it, in the directory `test/data/`, are additional DataLad subdatasets that contain the data he used for testing. Lastly, the DataLad superdataset contains a LaTeX `.tex` file with the text of the manuscript. When everything is set up, a single command line call triggers (optional) data retrieval from GitHub repositories of the datasets, computation of results and figures, automatic embedding of results and figures into his manuscript upon computation, and PDF compiling. When he notices the error in his script, his manuscript is recompiled and updated with a single command line call, and when he learns about the data error, he updates the respective DataLad dataset to the fixed state while preserving the history of the data repository.

He makes his superdataset a public repository on GitHub, and anyone who clones it can obtain the data automatically and recompute and recompile the full manuscript with all results. Steve never had more confidence in his research results and proudly submits his manuscript. During review, the color scheme update in his algorithm sourcecode is integrated with a simple update of the `algorithm/` subdataset, and upon command-line invocation his manuscript updates itself with the new figures.

> ⚠️ **Take a look at the real manuscript dataset**
>
> The actual manuscript this use case is based on can be found here[566]: https://github.com /psychoinformatics-de/paper-remodnav/. **datalad clone** the repository and follow the few instructions in the README to experience the DataLad approach described above. There is also a slimmed down template that uses the analysis demonstrated in *YODA-compliant data analysis projects* (page 143) and packages it up into a reproducible paper using the same tools: github.com/datalad-handbook/repro-paper-sketch/[567].
>
> ---
> [566] https://github.com/psychoinformatics-de/paper-remodnav/
> [567] https://github.com/datalad-handbook/repro-paper-sketch/

23.3 Step-by-Step

datalad create a DataLad dataset. In this example, it is named "algorithm-paper", and **datalad create** uses the yoda procedure[573] to apply useful configurations for a data analysis project:

```
$ datalad create -c yoda algorithm-paper

[INFO   ] Creating a new annex repo at /home/adina/repos/testing/algorithm-paper
create(ok): /home/adina/repos/testing/algorithm-paper (dataset)
```

This newly created directory already has a `code/` directory that will be tracked with Git and some `README.md` and `CHANGELOG.md` files thanks to the yoda procedure applied above. Additionally, create a subdirectory `data/` within the dataset. This project thus already has a comprehensible structure:

```
$ cd algorithm-paper
$ mkdir data

# You can checkout the directory structure with the tree command

$ tree
algorithm-paper
├── CHANGELOG.md
├── code
│   └── README.md
├── data
└── README.md
```

All of your analyses scripts should live in the `code/` directory, and all input data should live in the `data/` directory.

To populate the DataLad dataset, add all the data collections you want to perform analyses on as individual DataLad subdatasets within `data/`. In this example, all data collections are already DataLad datasets or git repositories and hosted on GitHub. **datalad clone** therefore installs them as subdatasets, with `-d ../` registering them as subdatasets to the superdataset[574].

```
$ cd data
# clone existing git repositories with data (-s specifies the source, in this case,␣
↪GitHub repositories)
# -d points to the root of the superdataset
datalad clone -d ../ https://github.com/psychoinformatics-de/studyforrest-data-phase2.git

[INFO   ] Cloning https://github.com/psychoinformatics-de/studyforrest-data-phase2.git [1␣
↪other candidates] into '/home/adina/repos/testing/algorithm-paper/data/raw_eyegaze'
install(ok): /home/adina/repos/testing/algorithm-paper/data/raw_eyegaze (dataset)

$ datalad clone -d ../ git@github.com:psychoinformatics-de/studyforrest-data-
↪eyemovementlabels.git

[INFO   ] Cloning git@github.com:psychoinformatics-de/studyforrest-data-eyemovementlabels.
↪git into '/home/adina/repos/testing/algorithm-paper/data/studyforrest-data-
↪eyemovementlabels'
Cloning (compressing objects):  45% 1.80k/4.00k [00:01<00:01, 1.29k objects/s
[...]
```

Any script we need for the analysis should live inside `code/`. During script writing, save any changes to you want to record in your history with **datalad save**.

The eventual outcome of this work is a GitHub repository that anyone can use to get the data and recompute all results when running the script after cloning and setting up the necessary

---

[574] You can read up on cloning datasets as subdatasets again in section *Install datasets* (page 46).

software. This requires minor preparation:

- The final analysis should be able to run on anyone's filesystem. It is therefore important to reference datafiles with the scripts in code/ as RELATIVE PATHs instead of hard-coding ABSOLUTE PATHs.

- After cloning the algorithm-paper repository, data files are not yet present locally. To spare users the work of a manual **datalad get**, you can have your script take care of data retrieval via DataLad's Python API.

These two preparations can be seen in this excerpt from the Python script:

```python
# import DataLad's API
from datalad.api import get

# note that the datapath is relative
datapath = op.join('data',
                   'studyforrest-data-eyemovementlabels',
                   'sub*',
                   '*run-2*.tsv')
data = sorted(glob(datapath))

# this will get the data if it is not yet retrieved
get(dataset='.', path=data)
```

Lastly, **datalad clone** the software repository as a subdataset in the root of the superdataset[575].

```
# in the root of ``algorithm-paper`` run
$ datalad clone -d . git@github.com:psychoinformatics-de/remodnav.git
```

This repository has also subdatasets in which the datasets used for testing live (tests/data/):

```
$ tree
[...]

|   ├── remodnav
|   ├── clf.py
|   ├── __init__.py
|   ├── __main__.py
|   └── tests
|       ├── data
|       │   ├── anderson_etal
|       │   └── studyforrest
```

At this stage, a public algorithm-paper repository shares code and data, and changes to any dataset can easily be handled by updating the respective subdataset. This already is a big leap towards open and reproducible science. Thanks to DataLad, code, data, and the history of all code and data are easily shared - with exact versions of all components and bound together in a single, fully tracked research object. By making use of the Python API of DataLad and RELATIVE PATHs in scripts, data retrieval is automated, and scripts can run on any other computer.

---

[575] Note that the software repository may just as well be cloned into data/.

23.4 Automation with existing tools

To go beyond that and include freshly computed results in a manuscript on the fly does not require DataLad anymore, only some understanding of Python, LaTeX, and Makefiles. As with most things, its a surprisingly simple challenge if one has just seen how to do it once. This last section will therefore outline how to compile the results into a PDF manuscript and automate this process. In principle, the challenge boils down to:

1. have the script output results (only requires print() statements)

2. capture these results automatically (done with a single line of Unix commands)

3. embed the captured results in the PDF (done with one line in the .tex file and some clever referencing)

4. automate as much as possible to keep it as simple as possible (done with a Makefile)

That does not sound too bad, does it? Let's start by revealing how this magic trick works. Everything relies on printing the results in the form of user-defined LaTeX definitions (using the \newcommand command), referencing those definitions in your manuscript where the results should end up, and bind the \newcommands as \input{} to your .tex file. But lets get there in small steps.

First, if you want to read up on the \newcommand, please see its documentation[568]. The command syntax looks like this:

\newcommand{\name}[num]{definition}

What we want to do, expressed in the most human-readable form, is this:

\newcommand{\Table1Cell1Row1}{0.67}

where 0.67 would be a single result computed by your script. This requires print() statements that look like this in the most simple form (excerpt from script):

```python
print('\\newcommand{\\maxmclf}{{%.2f}}' % max_mclf)
```

where max_mclf is a variable that stores the value of one computation.

Tables and references to results within the .tex files then do not contain the specific value 0.67 (this value would change if the data changes, or other parameters), but \maxmclf (and similar, unique names for other results). For full tables, one can come up with naming schemes that make it easy to fill tables with unique names with minimal work, for example like this (excerpt):

```latex
\begin{table}[tbp]
  \caption{Cohen's Kappa reliability between human coders (MN, RA),
  and \remodnav\ (AL) with each of the human coders.
  }
  \label{tab:kappa}
  \begin{tabular*}{0.5\textwidth}{c @{\extracolsep{\fill}}llll}
    \textbf {Fixations} &                  &                  \\
    \hline\noalign{\smallskip}
    Comparison          & Images           & Dots             \\
    \noalign{\smallskip}\hline\noalign{\smallskip}
    MN versus RA        & \kappaRAMNimgFix & \kappaRAMNdotsFix \\
    AL versus RA        & \kappaALRAimgFix & \kappaALRAdotsFix \\
```

(continues on next page)

---

[568] https://en.wikibooks.org/wiki/LaTeX/Macros

```
AL versus MN         & \kappaALMNimgFix & \kappaALMNdotsFix \\
\noalign{\smallskip}
\textbf{Saccades}    &                  &                  \\
\hline\noalign{\smallskip}
Comparison           & Images           & Dots             \\
\noalign{\smallskip}\hline\noalign{\smallskip}
MN versus RA         & \kappaRAMNimgSac & \kappaRAMNdotsSac \\
AL versus RA         & \kappaALRAimgSac & \kappaALRAdotsSac \\
AL versus MN         & \kappaALMNimgSac & \kappaALMNdotsSac \\
\noalign{\smallskip}
% [..] more content omitted
  \end{tabular*}
\end{table}
```

Without diving into the context of the paper, this table contains results for three three comparisons ("MN versus RA", "AL versus RA", "AL versus MN"), for three event types (Fixations, Saccades, and post-saccadic oscillations (PSO)), and three different stimulus types (Images, Dots, and Videos). The latter event and stimulus are omitted for better readability of the `.tex` excerpt. Here is how this table looks like in the manuscript (cropped to match the `.tex` snippet):

Fixations		
Comparison	Images	Dots
MN versus RA	0.84	0.65
AL versus RA	0.55	0.37
AL versus MN	0.52	0.45
Saccades		
Comparison	Images	Dots
MN versus RA	0.91	0.81
AL versus RA	0.78	0.72
AL versus MN	0.78	0.78

[...]

[...]

It might appear tedious to write scripts that output results for such tables with individual names. However, `print()` statements to fill those tables can utilize Pythons string concatenation methods and loops to keep the code within a few lines for a full table, such as

```python
# iterate over stimulus categories
for stim in ['img', 'dots', 'video']:
    # iterate over event categories
    for ev in ['Fix', 'Sac', 'PSO']:

    [...]

        # create the combinations
        for rating, comb in [('RAMN', [RA_res_flat, MN_res_flat]),
                             ('ALRA', [RA_res_flat, AL_res_flat]),
                             ('ALMN', [MN_res_flat, AL_res_flat])]:
            kappa = cohen_kappa_score(comb[0], comb[1])
            label = 'kappa{}{}{}'.format(rating, stim, ev)
            # print the result
            print('\\newcommand{\\%s}{%s}' % (label, '%.2f' % kappa))
```

Running the python script will hence print plenty of LaTeX commands to your screen (try it out in the actual manuscript, if you want!). This was step number 1 of 4.

> 💡 **M23.1 How about figures?**
>
> To include figures, the figures just need to be saved into a dedicated location (for example a directory `img/`) and included into the `.tex` file with standard LaTeX syntax. Larger figures with subfigures can be created by combining several figures:

```
\begin{figure*}[tbp]
  \includegraphics[trim=0 8mm 3mm 0,clip,width=.5\textwidth]{img/mainseq_lab}
  \includegraphics[trim=8mm 8mm 0 0,clip,width=.5\textwidth-3.3mm]{img/mainseq_sub_
↪lab} \\
  \includegraphics[trim=0 0 3mm 0,clip,width=.5\textwidth]{img/mainseq_mri}
  \includegraphics[trim=8mm 0 0 0,clip,width=.5\textwidth-3.3mm]{img/mainseq_sub_
↪mri}

  \caption{Main sequence of eye movement events during one 15 minute sequence of
  the movie (segment 2) for lab (top), and MRI participants (bottom). Data
  across all participants per dataset is shown on the left, and data for a single
  exemplary participant on the right.}

  \label{fig:overallComp}
\end{figure*}
```

This figure looks like this in the manuscript:

Fig. 6 Main sequence of eye movement events during one 15 minute sequence of the movie (segment 2) for lab (top), and MRI participants (bottom). Data across all participants per dataset is shown on the left, and data for a single exemplary participant on the right.
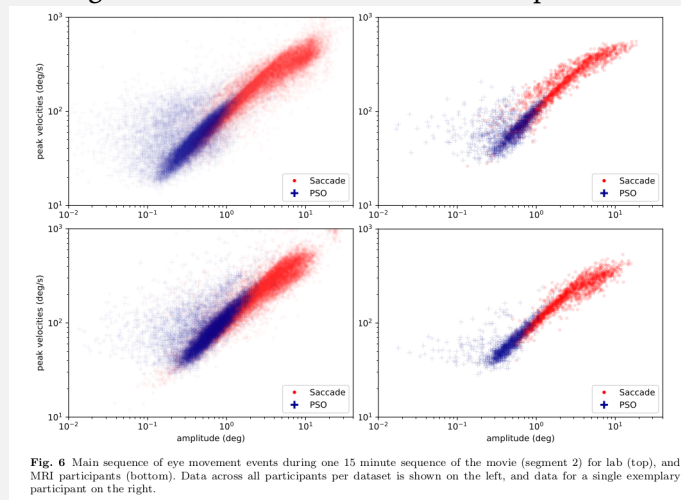
For step 2 and 3, the print statements need to be captured and bound to the .tex file. The tee[569] command can write all of the output to a file (called results_def.tex):

```
code/mk_figuresnstats.py -s | tee results_def.tex
```

This will redirect every print statement the script wrote to the terminal into a file called results_def.tex. This file will hence be full of \newcommand definitions that contain the results of the computations.

For step 3, one can include this file as an input source into the .tex file with

```
\begin{document}
\input{results_def.tex}
```

Upon compilation of the .tex file into a PDF, the results of the computations captured with \newcommand definitions are inserted into the respective part of the manuscript.

The last step is to automate this procedure. So far, the script would need to be executed with a command line call, and the PDF compilation would require another commandline call. One way to automate this process are Makefiles[570]. make is a decades-old tool known to many and

---

[569] https://en.wikipedia.org/wiki/Tee_(command)
[570] https://en.wikipedia.org/wiki/Make_(software)

bears the important advantage that is will deliver results regardless of what actually needs to be done with a single make call – whether it is executing a Python script, running bash commands, or rendering figures, or all of this. Here is the one used for the manuscript:

```
1   all: main.pdf
2
3   main.pdf: main.tex tools.bib EyeGaze.bib results_def.tex figures
4       latexmk -pdf -g $<
5
6   results_def.tex: code/mk_figuresnstats.py
7       bash -c 'set -o pipefail; code/mk_figuresnstats.py -s | tee results_def.tex'
8
9   figures: figures-stamp
10
11  figures-stamp: code/mk_figuresnstats.py
12      code/mk_figuresnstats.py -f -r -m
13      $(MAKE) -C img
14      touch $@
15
16  clean:
17      rm -f main.bbl main.aux main.blg main.log main.out main.pdf main.tdo main.fls main.
    ↪fdb_latexmk example.eps img/*eps-converted-to.pdf texput.log results_def.tex figures-
    ↪stamp
18      $(MAKE) -C img clean
```

One can read a Makefile as a recipe:

- Line 1: "The overall target should be main.pdf (the final PDF of the manuscript)."

- Line 2-3: "To make the target main.pdf, the following files are required: main.tex (the manuscript's .tex file), tools.bib & EyeGaze.bib (bibliography files), results_def.tex (the results definitions), and figures (a section not covered here, about rendering figures with inkscape prior to including them in the manuscript). If all of these files are present, the target main.pdf can be made by running the command latexmk -pdf -g"

- Line 5-6: "To make the target results_def.tex, the script code/mk_figuresnstats.py is required. If the file is present, the target results_def.tex can be made by running the command bash -c 'set -o pipefail; code/mk_figuresnstats.py -s | tee results_def.tex'"

This triggers the execution of the script, collection of results in results_def.tex, and PDF compilation upon typing make. The last three lines define that a make clean removes all computed files, and also all images.

Finally, by wrapping make in a **datalad run** command, the computation of results and compiling of the manuscript with all generated output can be written to the history of the superdataset. datalad run make will thus capture all provenance for the results and the final PDF.

Thus, by using DataLad and its Python API, a few clever Unix and LaTeX tricks, and Makefiles, anyone can create a reproducible paper. This saves time, increases your own trust in the results, and helps to make a more convincing case with your research. If you have not yet, but are curious, checkout the manuscript this use case is based on[571]. Any questions can be asked by opening an issue[572].

---

[571] http://github.com/psychoinformatics-de/paper-remodnav/

[572] https://github.com/psychoinformatics-de/paper-remodnav/issues/new

STUDENT SUPERVISION IN A RESEARCH PROJECT

This use case will demonstrate a workflow that uses DataLad tools and principles to assist in technical aspects of supervising research projects with computational components. It demonstrates how a DataLad dataset comes with advantages that mitigate technical complexities for trainees and allows high-quality supervision from afar with minimal effort and time commitment from busy supervisors. It furthermore serves to log undertaken steps, establishes trust in an analysis, and eases collaboration.

Successful workflows rely on more knowledgeable "trainers" (i.e., supervisors, or a more experienced collaborator) for a quick initial dataset setup with optimal configuration, and an introduction to the YODA principles and basic DataLad commands. Subsequently, supervision and collaboration is made easy by the distributed nature of a dataset. Afterwards, reuse of a students work is made possible by the modular nature of the dataset. Students can concentrate on questions relevant for the field and research topic, and computational complexities are minimized.

24.1 The Challenge

Megan is a graduate student and does an internship in a lab at a partnering research institution. As she already has experience in data analysis, and the time of her supervisor is limited, she is given a research question to work on autonomously. The data are already collected, and everyone involved is certain that Megan will be fine performing the analyses she has experience with. Her supervisor confidently proposes the research project as a conference talk Megan should give at the end of her stay. Megan is excited about the responsibility and her project, and can not wait to start.

On the first day, her supervisor spends an hour to show her the office, the coffee machine, and they chat about the high-level aspects of the projects: Which is the relevant literature, who collected the data, how long should the final talk be. Megan has many procedural questions, but the hour is over fast, and it is difficult to find time to meet again. As it turns out, her supervisor will leave the country for a three month visit to a lab in Japan soon, and is very busy preparing this stay and coordinating other projects. However, everyone is confident that Megan will be just fine. The IT office issues an account on the computational cluster for her, and the postdoc that collected the data points her to the directories in which the data are stored.

When she starts, Megan realizes that she has no experience with the Linux-based operating system running on the compute cluster. She knows very well how to write scripts to perform very complex analyses, but needs to invest much time to understand basic concepts and relevant commands on the cluster because no-one is around to give her a quick introduction. When she starts her computations, she accidentally overwrites a data file in the data collection, and emails

the postdoc for help. He luckily has a backup of the data and is able to restore the original state, but grimly CCs her supervisor in his response email to her. Not being told where to store analysis results in, Megan saves the results in a not backed-up `scratch` directory. With ambiguous, hard-to-make-sense-of emails her supervisor sends at 3am, Megan tries to comply to the instructions she extracts from the emails, and reports back lengthy explanations of what she is doing that her supervisor rarely has time to read. Without an interactive discussion or feedback component, Megan is very unsure about what she is supposed to do, and saves multiple different analysis scripts and results of them inside of the scratch folder.

When her supervisor returns and meets for a project update, he scolds her for the bad organization, and the no-backup storage choice. With a pressing timeline, Megan is told to write down her results. She is discouraged when she finally gets feedback on them and learns that she interpreted one instruction of her supervisor differently from what was meant by it, deeming all of her results irrelevant. Not trusting Megan's analyses anymore, her supervisor cancels the talk and has the postdoc take over. Megan feels incompetent and regards the stay as a waste of time, her supervisor is unhappy about the mis-communication and lack of results, and the postdoc taking over is unable to comprehend what was done so far and needs to start over new, even though all analysis scripts were correct and very relevant for the future of the project.

24.2 The DataLad Approach

When Megan arrives in the lab, her supervisor and the postdoc that collected the data take an hour to meet and talk about the upcoming project. To ease the technical complexities for a new student like Megan on an unfamiliar computational infrastructure, they talk about the YODA principles, basic DataLad commands, and set up a project dataset for Megan to work in. Inside of this dataset, the original data are cloned as a subdataset, code is tracked with Git, and the appropriate software is provided with a containerized image tracked in the dataset. Megan can adopt the version control workflow and data analysis principles very fast and is thankful for the brief but sufficient introduction. When her supervisor leaves for Japan, they stay in touch via email, but her supervisor also checks the development of the project and occasionally skims through Megan's code updates from afar every other week. When he notices that one of his instructions was ambiguous and Megan's approach to it misguided, he can intervene right away. Megan feels comfortable and confident that she is doing something useful and learns a lot about data management in the safe space of a version controlled dataset. Her supervisor can see how well made Megan's analysis methods are, and has trust in her results. Megan proudly presents the results of her analysis and leaves with many good experiences and lots of new knowledge. Her supervisor is happy about the progress done on the project, and the dataset is a standalone "lab-notebook" that anyone can later use as a detailed log to make sense of what was done. As an ongoing collaboration, Megan, the postdoc, and her supervisor write up a paper on the analysis and use the analysis dataset as a subdataset in this project.

24.3 Step-by-Step

Megan's supervisor is excited that she comes to visit the lab and trusts her to be a diligent, organized, and capable researcher. But he also does not have much time for a lengthy introduction to technical aspects unrelated to the project, interactive teaching, or in-person supervision. Megan in turn is a competent student and eager to learn new things, but she does not have experience with DataLad, version control, or the computational cluster.

As a first step, therefore, her supervisor and the postdoc prepare a preconfigured dataset in a dedicated directory everyone involved in the project has access to:

```
$ datalad create -c yoda project-megan
```

All data that this lab generates or uses is a standalone DataLad dataset that lives in a dedicated data\ directory on a server. To give Megan access to the data without endangering or potentially modifying the pristine data kept in there, complying to the YODA principles, they clone the data she is supposed to analyze as a subdataset:

```
$ cd project-megan
$ datalad clone -d . \
  /home/data/ABC-project \
  data/ABC-project

[INFO   ] Cloning /home/data/ABC-project [1 other candidates] into '/home/projects/
↪project-megan/data/ABC-project'
[INFO   ] Remote origin not usable by git-annex; setting annex-ignore
install(ok): data/ABC-project (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

The YODA principle and the data installation created a comprehensive directory structure and configured the code\ directory to be tracked in Git, to allow for easy, version-controlled modifications without the necessity to learn about locked content in the annex.

```
$ tree
.
├── CHANGELOG.md
├── code
│   └── README.md
├── data
│   └── ABC-project [13 entries exceeds filelimit, not opening dir]
└── README.md
```

Within a 20-minute walk-through, Megan learns the general concepts of version- control, gets an overview of the YODA principles[577], configures her Git identity with the help of her supervisor, and is given an introduction to the most important DataLad commands relevant to her, **datalad save**[578], **datalad containers-run**[579], and **datalad rerun**[580]. For reference, they also give her the *cheat sheet* (page 496) and the link to the DataLad handbook as a resource if she has further questions.

To make the analysis reproducible, they spent the final part of the meeting on adding the labs default singularity image to the dataset. The lab has a singularity image with all the relevant software on Singularity-Hub[576], and it can easily be added to the dataset with the DataLad-containers extension[579]:

---

[577] Find out more about the YODA principles in section *YODA: Best practices for data analyses in a dataset* (page 136)
[578] Find out more about datalad save in section *Modify content* (page 43)
[579] Find out more about the `datalad containers` extension in section TODO:link once it exists
[580] Find out more about the `datalad rerun` command in section *DataLad, Re-Run!* (page 64)
[576] https://singularity-hub.org/

```
$ datalad containers-add somelabsoftware --url shub://somelab/somelab-
↪container:Softwaresetup
```

With the container image registered in the dataset, Megan can perform her analysis in the correct software environment, does not need to setup software herself, and creates a more reproducible analysis.

With only a single command to run, Megan finds it easy to version control her scripts and gets into the habit of running **datalad save** frequently. This way, she can fully concentrate on writing up the analysis. In the beginning, her commit messages may not be optimal, and the changes she commits into a single commit might have better been split up into separate commits. But from the very beginning she is able to version control her progress, and she gets more and more proficient as the project develops.

Knowing the YODA principles gives her clear and easy-to-follow guidelines on how to work. Her scripts are producing results in dedicated output/ directories and are executed with **datalad containers-run** to capture the provenance of how which result came to be with which software. These guidelines are not complex, and yet make her whole workflow much more comprehensible, organized, and transparent.

The preconfigured DataLad dataset thus minimized the visible technical complexity. Just a few commands and standards have a large positive impact on her project and Megan learns these new skills fast. It did not take her supervisor much time to configure the dataset or give her an introduction to the relevant commands, and yet it ensured her to be able to productively work and contribute her expertise to the project.

Her supervisor can also check how the project develops if Megan asks for assistance or if he is curious – even from afar and whenever he has some 15 minutes of spare-time. When he notices that Megan must have misunderstood one of his emails, he can intervene and contact Megan by their preferred method of communication, and/or push a fix or comment to the project, as he has write-access. This enables him to stay up-to-date independent of emails or meetings with Megan, and to help when necessary without much trouble. When they talk, they focus on the code and analysis at hand, and not solely on verbal reports.

Megan finishes her analysis well ahead of time and can prepare her talk. Together with her supervisor she decides which figures look good and which results are important. All results that are deemed irrelevant can be dropped to keep the dataset lean, but could be recomputed as their provenance was tracked. Finally, the data analysis project is cloned as an input into a new dataset created for collaborative paper-writing on the analysis:

```
$ datalad create megans-paper
$ cd megans-paper
$ datalad clone -d . \
  /home/projects/project-megan \
  analysis

[INFO   ] Cloning /home/projects/project-megan [1 other candidates] into '/home/paper/
↪megans-paper'
[INFO   ]    Remote origin not usable by git-annex; setting annex-ignore
install(ok): analysis (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

Even as Megan returns to her home institution, they can write up the paper on her analysis collaboratively, and her co-authors have a detailed research log of the project within the dataset's history.

In summary, DataLad can help to effectively manage student supervision in computational projects. It requires minimal effort, but comes with great benefit:

- Appropriate data management is made a key element of the project and handled from the start, not an afterthought that needs to be addressed at the end of its lifetime.

- The dataset becomes the lab notebook, hence a valid and detailed log is always available and accessible to supervisor and trainee.

- supervisors can efficiently prepare for meetings in a way that does not rely exclusively on a students report. This shifts the focus from trust in a student to trust in a student's work.

- supervisors can provide feedback, not only high-level based on a presentation, but much more detailed, and also on process aspects if desired/necessary: Supervisors can directly contribute in a way that is as auditable/accountable as the student's own contributions – for both parties the strict separation and tracking of any external inputs of a project make it possible (when a project is completed) that a supervisor can efficiently test the integrity of the inputs, discard them (if unmodified), and only archive the outputs that are unique to the project – which then can become a modular component for re-use in a future project.

A BASIC AUTOMATICALLY AND COMPUTATIONALLY REPRODUCIBLE NEUROIMAGING ANALYSIS

This use case sketches the basics of a portable analysis of public neuroimaging data that can be automatically computationally reproduced by anyone:

1. Public open data stems from THE DATALAD SUPERDATASET ///.

2. Automatic data retrieval can be ensured by using DataLad's commands in the analysis scripts, or the `--input` specification of **datalad run**,

3. Analyses are executed using **datalad run** and **datalad rerun** commands to capture everything relevant to reproduce the analysis.

4. The final dataset can be kept as lightweight as possible by dropping input that can be easily re-obtained.

5. A complete reproduction of the computation (including input retrieval), is possible with a single **datalad rerun** command.

This use case is a specialization of *Writing a reproducible paper* (page 399), and a simpler version of *An automatically and computationally reproducible neuroimaging analysis from scratch* (page 420): It is a data analysis that requires and creates large data files, uses specialized analysis software, and is fully automated using solely DataLad commands and tools. While exact data types, analysis methods, and software mentioned in this use case belong to the scientific field of neuroimaging, the basic workflow is domain-agnostic.

25.1 The Challenge

Creating reproducible (scientific) analyses seems to require so much: One needs to share data, scripts, results, and instructions on how to use data and scripts to obtain the results. A researcher at any stage of their career can struggle to remember which script needs to be run in which order, or to create comprehensible instructions for others on where and how to obtain data and how to run which script at what point in time. This leads to failed replications, a loss of confidence in results, and major time requirements for anyone trying to reproduce others or even their own analyses.

25.2 The DataLad Approach

Scientific studies should be reproducible, and with the increasing accessibility of data, there is not much excuse for a lack of reproducibility anymore. DataLad can help with the technical

aspects of reproducible science.

For neuroscientific studies, THE DATALAD SUPERDATASET /// provides unified access to a large amount of data. Using it to install datasets into an analysis-superdataset makes it easy to share this data together with the analysis. By ensuring that all relevant data is downloaded via **datalad get** via DataLad's command line tools in the analysis scripts, or `--input` specifications in a **datalad run**, an analysis can retrieve all required inputs fully automatically during execution. Recording executed commands with **datalad run** allows to rerun complete analysis workflows with a single command, even if input data does not exist locally. Combining these three steps allows to share fully automatically reproducible analyses as lightweight datasets.

25.3 Step-by-Step

It always starts with a dataset:

```
$ datalad create -c yoda demo
[INFO] Creating a new annex repo at /home/me/usecases/repro/demo
[INFO] Scanning for unlocked files (this may take some time)
[INFO] Running procedure cfg_yoda
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
create(ok): /home/me/usecases/repro/demo (dataset)
```

For this demo we are using two public brain imaging datasets that were published on OpenFMRI.org[581], and are available from THE DATALAD SUPERDATASET /// (datasets.datalad.org). When installing datasets from this superdataset, we can use its abbreviation ///. The two datasets, ds000001[582] and ds000002[583], are installed into the subdirectory inputs/.

```
$ cd demo
$ datalad clone -d . ///openfmri/ds000001 inputs/ds000001
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro/demo/inputs/ds000001)
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000001 to /home/me/
↪usecases/repro/demo/inputs/ds000001
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000001/.git to /
↪home/me/usecases/repro/demo/inputs/ds000001
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro/demo/inputs/ds000001)
[INFO] scanning for unlocked files (this may take some time)
install(ok): inputs/ds000001 (dataset)
add(ok): inputs/ds000001 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
```

(continues on next page)

---

[581] https://legacy.openfmri.org/
[582] https://legacy.openfmri.org/dataset/ds000001/
[583] https://legacy.openfmri.org/dataset/ds000002/

```
  add (ok: 3)
  install (ok: 1)
  save (ok: 2)


$ cd demo
$ datalad clone -d . ///openfmri/ds000002 inputs/ds000002
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro/demo/inputs/ds000002)
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000002 to /home/me/
→usecases/repro/demo/inputs/ds000002
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000002/.git to /
→home/me/usecases/repro/demo/inputs/ds000002
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro/demo/inputs/ds000002)
[INFO] scanning for unlocked files (this may take some time)
install(ok): inputs/ds000002 (dataset)
add(ok): inputs/ds000002 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  install (ok: 1)
  save (ok: 2)
```

Both datasets are now registered as subdatasets, and their precise versions (e.g. in the form of
the commit shasum of the latest commit) are on record:

```
$ datalad --output-format '{path}: {gitshasum}' subdatasets
/home/me/usecases/repro/demo/inputs/ds000001: f7fe2e38852915e7042ca1755775fcad0ff166e5
/home/me/usecases/repro/demo/inputs/ds000002: 6b16eff0c9e8d443ee551784981ddd954f657071
```

DataLad datasets are fairly lightweight in size, they only contain pointers to data and history
information in their minimal form. Thus, so far very little data were actually downloaded:

```
$ du -sh inputs/
15M        inputs/
```

Both datasets would actually be several gigabytes in size, once the dataset content gets down-
loaded:

```
$ datalad -C inputs/ds000001 status --annex
$ datalad -C inputs/ds000002 status --annex
130 annex'd files (2.3 GB recorded total size)
nothing to save, working tree clean
274 annex'd files (2.7 GB recorded total size)
nothing to save, working tree clean
```

Both datasets contain brain imaging data, and are compliant with the BIDS standard[584]. This
makes it really easy to locate particular images and perform analysis across datasets.

---

[584] https://bids.neuroimaging.io/

Here we will use a small script that performs 'brain extraction' using FSL[585] as a stand-in for a full analysis pipeline. The script will be stored inside of the code/ directory that the yoda-procedure created that at the time of dataset-creation.

```
$ cat << EOT > code/brain_extraction.sh
# enable FSL
. /etc/fsl/5.0/fsl.sh

# obtain all inputs
datalad get \$@
# perform brain extraction
count=1
for nifti in \$@; do
  subdir="sub-\$(printf %03d \$count)"
  mkdir -p \$subdir
  echo "Processing \$nifti"
  bet \$nifti \$subdir/anat -m
  count=\$((count + 1))
done
EOT
```

Note that this script uses the **datalad get** command which automatically obtains the required files from their remote source – we will see this in action shortly.

We are saving this script in the dataset. This way, we will know exactly which code was used for the analysis. Everything inside of code/ is tracked with Git thanks to the yoda-procedure, so we can see more easily how it was edited over time. In addition, we will "tag" this state of the dataset with the tag setup_done to mark the repository state at which the analysis script was completed. This is optional, but it can help to identify important milestones more easily.

```
$ datalad save --version-tag setup_done -m "Brain extraction script" code/brain_
↪extraction.sh
add(ok): code/brain_extraction.sh (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Now we can run our analysis code to produce results. However, instead of running it directly, we will run it with DataLad – this will automatically create a record of exactly how this script was executed.

For this demo we will just run it on the structural images (T1w) of the first subject (sub-01) from each dataset. The uniform structure of the datasets makes this very easy. Of course we could run it on all subjects; we are simply saving some time for this demo. While the command runs, you should notice a few things:

1) We run this command with 'bash -e' to stop at any failure that may occur

2) You'll see the required data files being obtained as they are needed – and only those that are actually required will be downloaded (because of the appropriate --input specification of the **datalad run** – but as a **datalad get** is also included in the bash script, forgetting an --input specification would not be problem).

---

[585] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FSL

```
$ datalad run -m "run brain extract workflow" \
 --input "inputs/ds*/sub-01/anat/sub-01_T1w.nii.gz" \
 --output "sub-*/anat" \
 bash -e code/brain_extraction.sh inputs/ds*/sub-01/anat/sub-01_T1w.nii.gz
[INFO] Making sure inputs are available (this may take some time)
get(ok): inputs/ds000001/sub-01/anat/sub-01_T1w.nii.gz (file) [from web...]
get(ok): inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz (file) [from web...]
[INFO] == Command start (output follows) =====
action summary:
  get (notneeded: 4)
Processing inputs/ds000001/sub-01/anat/sub-01_T1w.nii.gz
Processing inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz
[INFO] == Command exit (modification check follows) =====
add(ok): sub-001/anat.nii.gz (file)
add(ok): sub-001/anat_mask.nii.gz (file)
add(ok): sub-002/anat.nii.gz (file)
add(ok): sub-002/anat_mask.nii.gz (file)
save(ok): . (dataset)
```

The analysis step is done, all generated results were saved in the dataset. All changes, including the command that caused them are on record:

```
$ git show --stat
commit 43927820508224c47b678b1f8f9cfca4f42f583c
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 10:28:12 2021 +0200

    [DATALAD RUNCMD] run brain extract workflow

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "bash -e code/brain_extraction.sh inputs/ds000001/sub-01/anat/sub-01_T1w.nii.
→gz inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz",
     "dsid": "b9aafaf5-0399-4726-805b-0440aa138be7",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [
      "inputs/ds*/sub-01/anat/sub-01_T1w.nii.gz"
     ],
     "outputs": [
      "sub-*/anat"
     ],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^

 sub-001/anat.nii.gz      | 1 +
 sub-001/anat_mask.nii.gz | 1 +
 sub-002/anat.nii.gz      | 1 +
 sub-002/anat_mask.nii.gz | 1 +
 4 files changed, 4 insertions(+)
```

DataLad has enough information stored to be able to re-run a command.

On command exit, it will inspect the results and save them again, but only if they are different. In our case, the re-run yields bit-identical results, hence nothing new is saved.

```
$ datalad rerun
[INFO] run commit 4392782; (run brain extract...)
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): sub-001/anat.nii.gz (file)
unlock(ok): sub-001/anat_mask.nii.gz (file)
unlock(ok): sub-002/anat.nii.gz (file)
unlock(ok): sub-002/anat_mask.nii.gz (file)
[INFO] == Command start (output follows) =====
action summary:
  get (notneeded: 4)
Processing inputs/ds000001/sub-01/anat/sub-01_T1w.nii.gz
Processing inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz
[INFO] == Command exit (modification check follows) =====
add(ok): sub-001/anat.nii.gz (file)
add(ok): sub-001/anat_mask.nii.gz (file)
add(ok): sub-002/anat.nii.gz (file)
add(ok): sub-002/anat_mask.nii.gz (file)
action summary:
  add (ok: 4)
  get (notneeded: 4)
  save (notneeded: 3)
  unlock (ok: 4)
```

Now that we are done, and have checked that we can reproduce the results ourselves, we can clean up. DataLad can easily verify if any part of our input dataset was modified since we configured our analysis, using **datalad diff** and the tag we provided:

```
$ datalad diff setup_done inputs
```

Nothing was changed.

With DataLad with don't have to keep those inputs around – without losing the ability to reproduce an analysis. Let's uninstall them, and check the size on disk before and after.

```
$ du -sh
26M          .
```

```
$ datalad uninstall inputs/*
drop(ok): /home/me/usecases/repro/demo/inputs/ds000001/sub-01/anat/sub-01_T1w.nii.gz␣
↪(file) [checking http://openneuro.s3.amazonaws.com/ds000001/ds000001_R1.1.0/
↪uncompressed/sub001/anatomy/highres001.nii.gz?
↪versionId=8TJ17W9WInNkQPdiQ9vS7wo8ZJ9llF80...]
drop(ok): inputs/ds000001 (directory)
uninstall(ok): inputs/ds000001 (dataset)
drop(ok): /home/me/usecases/repro/demo/inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz␣
↪(file) [checking http://openneuro.s3.amazonaws.com/ds000002/ds000002_R2.0.0/
↪uncompressed/sub-01/anat/sub-01_T1w.nii.gz?versionId=vXK2.bQ360phhPqbVV_n6RMYqaWAy4Dg...
↪]
drop(ok): inputs/ds000002 (directory)
uninstall(ok): inputs/ds000002 (dataset)
action summary:
  drop (ok: 4)
  uninstall (ok: 2)
```

```
$ du -sh
3.2M          .
```

The dataset is substantially smaller as all inputs are gone...

```
$ ls inputs/*
inputs/ds000001:

inputs/ds000002:
```

But as these inputs were registered in the dataset when we installed them, getting them back is very easy. Only the remaining data (our code and the results) need to be kept and require a backup for long term archival. Everything else can be re-obtained as needed, when needed.

As DataLad knows everything needed about the inputs, including where to get the right version, we can re-run the analysis with a single command. Watch how DataLad re-obtains all required data, re-runs the code, and checks that none of the results changed and need saving.

```
$ datalad rerun
[INFO] run commit 4392782; (run brain extract...)
[INFO] Making sure inputs are available (this may take some time)
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro/demo/inputs/ds000001)
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000001 to /home/me/
↪usecases/repro/demo/inputs/ds000001
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000001/.git to /
↪home/me/usecases/repro/demo/inputs/ds000001
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro/demo/inputs/ds000001)
[INFO] scanning for unlocked files (this may take some time)
install(ok): inputs/ds000001 (dataset) [Installed subdataset in order to get /home/me/
↪usecases/repro/demo/inputs/ds000001]
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro/demo/inputs/ds000002)
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000002 to /home/me/
↪usecases/repro/demo/inputs/ds000002
[INFO] Attempting to clone from http://datasets.datalad.org/openfmri/ds000002/.git to /
↪home/me/usecases/repro/demo/inputs/ds000002
[INFO] Start enumerating objects
[INFO] Start counting objects
[INFO] Start compressing objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro/demo/inputs/ds000002)
[INFO] scanning for unlocked files (this may take some time)
install(ok): inputs/ds000002 (dataset) [Installed subdataset in order to get /home/me/
↪usecases/repro/demo/inputs/ds000002]
get(ok): inputs/ds000001/sub-01/anat/sub-01_T1w.nii.gz (file) [from web...]
get(ok): inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz (file) [from web...]
unlock(ok): sub-001/anat.nii.gz (file)
unlock(ok): sub-001/anat_mask.nii.gz (file)
unlock(ok): sub-002/anat.nii.gz (file)
unlock(ok): sub-002/anat_mask.nii.gz (file)
[INFO] == Command start (output follows) =====
action summary:
  get (notneeded: 4)
Processing inputs/ds000001/sub-01/anat/sub-01_T1w.nii.gz
Processing inputs/ds000002/sub-01/anat/sub-01_T1w.nii.gz
```

(continues on next page)

```
[INFO] == Command exit (modification check follows) =====
add(ok): sub-001/anat.nii.gz (file)
add(ok): sub-001/anat_mask.nii.gz (file)
add(ok): sub-002/anat.nii.gz (file)
add(ok): sub-002/anat_mask.nii.gz (file)
action summary:
  add (ok: 4)
  get (notneeded: 2, ok: 2)
  install (ok: 2)
  save (notneeded: 3)
  unlock (ok: 4)
```

Reproduced!

This dataset could now be published and shared as a lightweight yet fully reproducible resource and enable anyone to replicate the exact same analysis – with a single command. Public data and reproducible execution for the win!

Note though that reproducibility can and should go further: With more complex software dependencies, it is inevitable to keep track of the software environment involved in the analysis as well. If you are curious on how to do this, read on into *An automatically and computationally reproducible neuroimaging analysis from scratch* (page 420).

AN AUTOMATICALLY AND COMPUTATIONALLY REPRODUCIBLE NEUROIMAGING ANALYSIS FROM SCRATCH

This use case sketches the basics of a portable analysis that can be automatically computationally reproduced, starting from the acquisition of a neuroimaging dataset with a magnetic resonance imaging (MRI) scanner up to complete data analysis results:

1. Two extension packages, datalad-container[586] and datalad-neuroimaging[587] extend DataLad's functionality with the ability to work with computational containers and neuroimaging data workflows.

2. The analysis is conducted in a way that leaves comprehensive provenance (including software environments) all the way from the raw data, and structures study components in a way that facilitates reuse.

3. It starts with preparing a raw data (dicom) dataset, and subsequently uses the prepared data for a general linear model (GLM) based analysis.

4. After completion, data and results are archived, and disk usage of the dataset is maximally reduced.

This use case is adapted from the ReproIn/DataLad tutorial[588] by Michael Hanke and Yaroslav Halchenko, given at the 2018 OHBM training course ran by ReproNim[589].

26.1 The Challenge

Allan is an exemplary neuroscientist and researcher. He has spent countless hours diligently learning not only the statistical methods for his research questions and the software tools for his computations, but also taught himself about version control and data standards in neuroimaging, such as the brain imaging data structure (BIDS[590]). For his final PhD project, he patiently acquires functional MRI data of many participants, and prepares it according to the BIDS standard afterwards. It takes him a full week of time and two failed attempts, but he eventually has a BIDS-compliant[591] dataset.

When he writes his analysis scripts he takes extra care to responsibly version control every change. He happily notices how much cleaner his directories are, and how he and others can transparently see how his code evolved. Once everything is set up, he runs his analysis using

---

[586] https://github.com/datalad/datalad-container
[587] https://github.com/datalad/datalad-neuroimaging
[588] http://www.repronim.org/ohbm2018-training/03-01-reproin/
[589] https://www.repronim.org/
[590] https://bids.neuroimaging.io/
[591] http://bids-standard.github.io/bids-validator/

large and complex neuroscientific software packages that he installed on his computer a few years back. Finally, he writes a paper and publishes his findings in a prestigious peer-reviewed journal. His data and code can be accessed by others easily, as he makes them publicly available. Colleagues and supervisors admire him for his wonderful contribution to open science.

However, a few months after publication, Allan starts to get emails by that report that his scripts do not produce the same results as the ones reported in the publication. Startled and confused he investigates what may be the issue. After many sleepless nights he realizes: The software he used was fairly old! More recent versions of the same software compute results slightly different, changed function's names, or fixed discovered bugs in the underlying source code. Shocked, he realizes that his scripts are even incompatible with the most recent release of the software package he used and throw an error. Luckily, he can quickly fix this by adding information about the required software versions to the README of his project, and he is grateful for colleagues and other scientists that provide adjusted versions of his code for more recent software releases. In the end, his results prove to be robust regardless of software version. But while Allen shared code and data, not including any information about his *software* environment prevented his analysis from becoming *computationally* reproducible.

26.2 The DataLad Approach

Even if an analysis workflow is fully captured and version-controlled, and data and code are being linked, an analysis may not reproduce. Comprehensive *computational* reproducibility requires that also the *software* involved in an analysis and its precise versions need to be known. DataLad can help with this. Using the `datalad-containers` extension, complete software environments can be captured in computational containers, added to (and thus shared together with) datasets, and linked with commands and outputs they were used for.

26.3 Step-by-Step

The first part of this Step-by-Step guide details how to computationally reproducibly and automatically reproducibly perform data preparation from raw DICOM[592] files to BIDS-compliant NifTi[593] images. The actual analysis, a first-level GLM for a localization task, is performed in the second part. A final paragraph shows how to prepare the dataset for the afterlife.

For this use case, two DataLad extensions are required:

- datalad-container[594] and
- datalad-neuroimaging[595]

You can install them via `pip` like this:

```
$ pip install datalad-neuroimaging datalad-container
```

Data Preparation

We start by creating a home for the raw data:

---

[592] https://www.dicomstandard.org/
[593] https://nifti.nimh.nih.gov/
[594] https://github.com/datalad/datalad-container
[595] https://github.com/datalad/datalad-neuroimaging

```
$ datalad create localizer_scans
$ cd localizer_scans
[INFO] Creating a new annex repo at /home/me/usecases/repro2/localizer_scans
[INFO] scanning for unlocked files (this may take some time)
create(ok): /home/me/usecases/repro2/localizer_scans (dataset)
```

For this example, we use a number of publicly available DICOM files. Luckily, at the time of data acquisition, these DICOMs were already equipped with the relevant metadata: Their headers contain all necessary information to identify the purpose of individual scans and encode essential properties to create a BIDS compliant dataset from them. The DICOMs are stored on Github (as a Git repository[601]), so they can be installed as a subdataset. As they are the raw inputs of the analysis, we store them in a directory we call inputs/raw.

```
$ datalad clone --dataset . \
 https://github.com/datalad/example-dicom-functional.git \
 inputs/rawdata
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro2/localizer_scans/inputs/rawdata)
[INFO] Attempting to clone from https://github.com/datalad/example-dicom-functional.git␣
↪to /home/me/usecases/repro2/localizer_scans/inputs/rawdata
[INFO] Start enumerating objects
[INFO] Start receiving objects
[INFO] Start resolving deltas
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro2/localizer_scans/
↪inputs/rawdata)
install(ok): inputs/rawdata (dataset)
add(ok): inputs/rawdata (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  install (ok: 1)
  save (ok: 2)
```

The **datalad subdatasets** reports the installed dataset to be indeed a subdataset of the super-dataset localizer_scans:

```
$ datalad subdatasets
subdataset(ok): inputs/rawdata (dataset)
```

Given that we have obtained *raw* data, this data is not yet ready for data analysis. Prior to performing actual computations, the data needs to be transformed into appropriate formats and standardized to an intuitive layout. For neuroimaging, a useful transformation is a transformation from DICOMs into the NifTi format, a format specifically designed for scientific analyses of brain images. An intuitive layout is the BIDS standard. Performing these transformations and standardizations, however, requires software. For the task at hand, HeudiConv[596], a DICOM converter, is our software of choice. Beyond converting DICOMs, it also provides assistance in converting a raw data set to the BIDS standard, and it integrates with DataLad to place con-

---

[601] "Why can such data exist as a Git repository, shouldn't large files be always stored outside of Git?" you may ask. The DICOMs exist in a Git-repository for a number of reasons: First, it makes them easily available for demonstrations and tutorials without involving DataLad at all. Second, the DICOMs are *comparatively* small: 21K per file. Importantly, the repository is not meant to version control those files *and* future states or derivatives and results obtained from them – this would bring a Git repositories to its knees.

[596] https://heudiconv.readthedocs.io/en/latest/

Chapter 26. An automatically and computationally reproducible neuroimaging analysis from scratch

verted and original data under Git/Git-annex version control, while automatically annotating files with sensitive information (e.g., non-defaced anatomicals, etc).

To take extra care to know exactly what software is used both to be able to go back to it at a later stage should we have the need to investigate an issue, and to capture *full* provenance of the transformation process, we are using a software container that contains the relevant software setup. A ready-made singularity[597] container is available from singularity-hub[598] at shub://ReproNim/ohbm2018-training:heudiconvn.

Using the **datalad containers-add** command we can add this container to the localizer_scans superdataset. We are giving it the name heudiconv.

```
$ datalad containers-add heudiconv --url shub://ReproNim/ohbm2018-training:heudiconvn
[INFO] Initiating special remote datalad
add(ok): .datalad/config (file)
save(ok): . (dataset)
containers_add(ok): /home/me/usecases/repro2/localizer_scans/.datalad/environments/
↪heudiconv/image (file)
action summary:
  add (ok: 1)
  containers_add (ok: 1)
  save (ok: 1)
```

The command **datalad containers-list** can verify that this worked:

```
$ datalad containers-list
heudiconv -> .datalad/environments/heudiconv/image
```

Great. The dataset now tracks all of the input data *and* the computational environment for the DICOM conversion. Thus far, we have a complete record of all components. Let's stay transparent, but also automatically reproducible in the actual data conversion by wrapping the necessary heudiconv command seen below:

```
$ heudiconv -f reproin -s 02 -c dcm2niix -b -l "" --minmeta -a . \
 -o /tmp/heudiconv.sub-02 --files inputs/rawdata/dicoms
```

within a **datalad containers-run** command. To save time, we will only transfer one subjects data (sub-02, hence the subject identifier -s 02 in the command). Note that the output below is how it indeed should look like – the software we are using in this example produces very wordy output.

```
$ datalad containers-run -m "Convert sub-02 DICOMs into BIDS" \
  --container-name heudiconv \
  heudiconv -f reproin -s 02 -c dcm2niix -b -l "" --minmeta -a . \
  -o /tmp/heudiconv.sub-02 --files inputs/rawdata/dicoms
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
INFO: Running heudiconv version 0.5.2-dev
INFO: Analyzing 5460 dicoms
INFO: Filtering out 0 dicoms based on their filename
WARNING: dcmstack without support of pydicom >= 1.0 is detected. Adding a plug
INFO: Generated sequence info for 1 studies with 1 entries total
INFO: Processing sequence infos to deduce study/session
```

(continues on next page)

---

[597] http://singularity.lbl.gov/
[598] https://singularity-hub.org/

```
INFO: Study session for {'locator': 'Hanke/Stadler/0083_transrep2', 'session': None,
↪'subject': '02'}
INFO: Need to process 1 study sessions
INFO: PROCESSING STARTS: {'outdir': '/tmp/heudiconv.sub-02/', 'session': None, 'subject':
↪'02'}
INFO: Processing 1 pre-sorted seqinfo entries
INFO: Processing 1 seqinfo entries
INFO: Doing conversion using dcm2niix
INFO: Converting ./sub-02/func/sub-02_task-oneback_run-01_bold (5460 DICOMs) -> ./sub-02/
↪func . Converter: dcm2niix . Output types: ('nii.gz', 'dicom')
INFO: Generating grammar tables from /usr/lib/python3.5/lib2to3/Grammar.txt
INFO: Generating grammar tables from /usr/lib/python3.5/lib2to3/PatternGrammar.txt
210729-10:15:13,343 nipype.workflow INFO:
         [Node] Setting-up "convert" in "/tmp/dcm2niixn_ufbfmu/convert".
INFO: [Node] Setting-up "convert" in "/tmp/dcm2niixn_ufbfmu/convert".
210729-10:15:14,429 nipype.workflow INFO:
         [Node] Running "convert" ("nipype.interfaces.dcm2nii.Dcm2niix"), a CommandLine␣
↪Interface with command:
dcm2niix -b y -z y -x n -t n -m n -f func -o . -s n -v n /tmp/dcm2niixn_ufbfmu/convert
INFO: [Node] Running "convert" ("nipype.interfaces.dcm2nii.Dcm2niix"), a CommandLine␣
↪Interface with command:
dcm2niix -b y -z y -x n -t n -m n -f func -o . -s n -v n /tmp/dcm2niixn_ufbfmu/convert
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127:Chris Rorden's dcm2niiX version v1.0.20180622␣
↪GCC6.3.0 (64-bit Linux)
INFO: stdout 2021-07-29T10:15:16.883127:Chris Rorden's dcm2niiX version v1.0.20180622␣
↪GCC6.3.0 (64-bit Linux)
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127:Found 5460 DICOM file(s)
INFO: stdout 2021-07-29T10:15:16.883127:Found 5460 DICOM file(s)
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127:swizzling 3rd and 4th dimensions (XYTZ ->␣
↪XYZT), assuming interslice distance is 3.300000
INFO: stdout 2021-07-29T10:15:16.883127:swizzling 3rd and 4th dimensions (XYTZ -> XYZT),␣
↪assuming interslice distance is 3.300000
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127:Warning: Images sorted by instance number ␣
↪[0020,0013](1..5460), but AcquisitionTime [0008,0032] suggests a different order␣
↪(160423..160223)
INFO: stdout 2021-07-29T10:15:16.883127:Warning: Images sorted by instance number  [0020,
↪0013](1..5460), but AcquisitionTime [0008,0032] suggests a different order (160423..
↪160223)
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127:Using RWVSlope:RWVIntercept = 4.00757:0
INFO: stdout 2021-07-29T10:15:16.883127:Using RWVSlope:RWVIntercept = 4.00757:0
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127: Philips Scaling Values RS:RI:SS = 4.00757:0:0.
↪0132383 (see PMC3998685)
INFO: stdout 2021-07-29T10:15:16.883127: Philips Scaling Values RS:RI:SS = 4.00757:0:0.
↪0132383 (see PMC3998685)
210729-10:15:16,883 nipype.interface INFO:
         stdout 2021-07-29T10:15:16.883127:Convert 5460 DICOM as ./func (80x80x35x156)
INFO: stdout 2021-07-29T10:15:16.883127:Convert 5460 DICOM as ./func (80x80x35x156)
210729-10:15:17,731 nipype.interface INFO:
         stdout 2021-07-29T10:15:17.731699:compress: "/usr/bin/pigz" -n -f -6 "./func.nii"
INFO: stdout 2021-07-29T10:15:17.731699:compress: "/usr/bin/pigz" -n -f -6 "./func.nii"
```

```
210729-10:15:17,731 nipype.interface INFO:
         stdout 2021-07-29T10:15:17.731699:Conversion required 3.229892 seconds (2.427477␣
→for core code).
INFO: stdout 2021-07-29T10:15:17.731699:Conversion required 3.229892 seconds (2.427477␣
→for core code).
210729-10:15:17,918 nipype.workflow INFO:
         [Node] Finished "convert".
INFO: [Node] Finished "convert".
INFO: Populating template files under ./
INFO: PROCESSING DONE: {'outdir': '/tmp/heudiconv.sub-02/', 'session': None, 'subject':␣
→'02'}
[INFO] == Command exit (modification check follows) =====
add(ok): CHANGES (file)
add(ok): README (file)
add(ok): dataset_description.json (file)
add(ok): participants.tsv (file)
add(ok): sourcedata/README (file)
add(ok): sourcedata/sub-02/func/sub-02_task-oneback_run-01_bold.dicom.tgz (file)
add(ok): sub-02/func/sub-02_task-oneback_run-01_bold.json (file)
add(ok): sub-02/func/sub-02_task-oneback_run-01_bold.nii.gz (file)
add(ok): sub-02/func/sub-02_task-oneback_run-01_events.tsv (file)
add(ok): sub-02/sub-02_scans.tsv (file)
add(ok): task-oneback_bold.json (file)
save(ok): . (dataset)
action summary:
  add (ok: 11)
  get (notneeded: 1)
  save (notneeded: 1, ok: 1)
```

Find out what changed after this command by comparing the most recent commit by DataLad (i.e., HEAD) to the previous one (i.e., HEAD~1) with **datalad diff**:

```
$ datalad diff -f HEAD~1
    added: CHANGES (file)
    added: README (file)
    added: dataset_description.json (file)
    added: participants.tsv (file)
    added: sourcedata/README (file)
    added: sourcedata/sub-02/func/sub-02_task-oneback_run-01_bold.dicom.tgz (file)
    added: sub-02/func/sub-02_task-oneback_run-01_bold.json (file)
    added: sub-02/func/sub-02_task-oneback_run-01_bold.nii.gz (file)
    added: sub-02/func/sub-02_task-oneback_run-01_events.tsv (file)
    added: sub-02/sub-02_scans.tsv (file)
    added: task-oneback_bold.json (file)
```

As expected, DICOM files of one subject were converted into NifTi files, **and** the outputs follow the BIDS standard's layout and naming conventions! But what's even better is that this action and the relevant software environment was fully recorded.

There is only one thing missing before the functional imaging data can be analyzed: A stimulation protocol, so that we know what stimulation was done at which point during the scan. Thankfully, the data was collected using an implementation that exported this information directly in the BIDS events.tsv format. The file came with our DICOM dataset and can be found at inputs/rawdata/events.tsv. All we need to do is copy it to the right location under the BIDS-mandated name. To keep track of where this file came from, we will also wrap the copying into a **datalad run** command. The {inputs} and {outputs} placeholders can help to avoid

duplication in the command call:

```
$ datalad run -m "Import stimulation events" \
  --input inputs/rawdata/events.tsv \
  --output sub-02/func/sub-02_task-oneback_run-01_events.tsv \
  cp {inputs} {outputs}
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): sub-02/func/sub-02_task-oneback_run-01_events.tsv (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): sub-02/func/sub-02_task-oneback_run-01_events.tsv (file)
save(ok): . (dataset)
```

`git log` shows what information DataLad captured about this command's execution:

```
$ git log -n 1
commit bbab0d13e6d827c2b2dd711386e86c3eab5e78a6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 10:15:38 2021 +0200

    [DATALAD RUNCMD] Import stimulation events

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "cp '{inputs}' '{outputs}'",
     "dsid": "cbaa0d64-00e2-4806-9fd4-d4738a889bf4",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [
       "inputs/rawdata/events.tsv"
     ],
     "outputs": [
       "sub-02/func/sub-02_task-oneback_run-01_events.tsv"
     ],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^
```

Analysis execution

Since the raw data are reproducibly prepared in BIDS standard we can now go further an conduct an analysis. For this example, we will implement a very basic first-level GLM analysis using FSL[599] that takes only a few minutes to run. As before, we will capture all provenance: inputs, computational environments, code, and outputs.

Following the YODA principles[602], the analysis is set up in a new dataset, with the input dataset `localizer_scans` as a subdataset:

```
# get out of localizer_scans
$ cd ../
```

(continues on next page)

---

[599] http://fsl.fmrib.ox.ac.uk/
[602] To re-read everything about the YODA principles, checkout out section *YODA: Best practices for data analyses in a dataset* (page 136).

```
$ datalad create glm_analysis
$ cd glm_analysis
[INFO] Creating a new annex repo at /home/me/usecases/repro2/glm_analysis
[INFO] scanning for unlocked files (this may take some time)
create(ok): /home/me/usecases/repro2/glm_analysis (dataset)
```

We install `localizer_scans` by providing its path as a `--source` to **datalad install**:

```
$ datalad clone -d . \
  ../localizer_scans \
  inputs/rawdata
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro2/glm_analysis/inputs/rawdata)
[INFO] Attempting to clone from ../localizer_scans to /home/me/usecases/repro2/glm_
↪analysis/inputs/rawdata
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro2/glm_analysis/inputs/
↪rawdata)
[INFO] scanning for unlocked files (this may take some time)
install(ok): inputs/rawdata (dataset)
add(ok): inputs/rawdata (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  install (ok: 1)
  save (ok: 2)
```

datalad subdatasets reports the number of installed subdatasets again:

```
$ datalad subdatasets
subdataset(ok): inputs/rawdata (dataset)
```

We almost forgot something really useful: Structuring the dataset with the help of DataLad! Luckily, procedures such as yoda can not only be applied upon creating of a dataset (as in *Create a dataset* (page 34)), but also with the **run-procedure** command (as in *Configurations to go* (page 126))

```
$ datalad run-procedure cfg_yoda
[INFO] Running procedure cfg_yoda
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
```

The analysis obviously needs custom code. For the simple GLM analysis with FSL we use:

1. A small script to convert BIDS-formatted `events.tsv` files into the `EV3` format FSL understands, available at https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/section23/scripts/events2ev3.sh

2. An FSL analysis configuration template script, available at https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/section23/scripts/ffa_design.fsf

These script should be stored and tracked inside the dataset within code/. The **datalad download-url** command downloads these scripts *and* records where they were obtained from:

```
$ datalad download-url  --path code/ \
  https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/section23/scripts/
↪events2ev3.sh \
  https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/section23/scripts/ffa_
↪design.fsf
[INFO] Downloading 'https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/
↪section23/scripts/events2ev3.sh' into '/home/me/usecases/repro2/glm_analysis/code/'
download_url(ok): /home/me/usecases/repro2/glm_analysis/code/events2ev3.sh (file)
[INFO] Downloading 'https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/
↪section23/scripts/ffa_design.fsf' into '/home/me/usecases/repro2/glm_analysis/code/'
download_url(ok): /home/me/usecases/repro2/glm_analysis/code/ffa_design.fsf (file)
add(ok): code/events2ev3.sh (file)
add(ok): code/ffa_design.fsf (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  download_url (ok: 2)
  save (ok: 1)
```

The commit message that DataLad created shows the URL where each script has been down-
loaded from:

```
$ git log -n 1
commit d0c450292c870c9eccea849d1cf0ae885136b438
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 10:15:44 2021 +0200

    [DATALAD] Download URLs

    URLs:
      https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/section23/scripts/
↪events2ev3.sh
      https://raw.githubusercontent.com/myyoda/ohbm2018-training/master/section23/scripts/
↪ffa_design.fsf
```

Prior to the actual analysis, we need to run the events2ev3.sh script to transform inputs into
the format that FSL expects. The **datalad run** makes this maximally reproducible and easy, as
the files given as --inputs and --outputs are automatically managed by DataLad.

```
$ datalad run -m 'Build FSL EV3 design files' \
  --input inputs/rawdata/sub-02/func/sub-02_task-oneback_run-01_events.tsv \
  --output 'sub-02/onsets' \
  bash code/events2ev3.sh sub-02 {inputs}
[INFO] Making sure inputs are available (this may take some time)
get(ok): inputs/rawdata/sub-02/func/sub-02_task-oneback_run-01_events.tsv (file) [from ↵
↪origin...]
[INFO] == Command start (output follows) =====
sub-02
1
[INFO] == Command exit (modification check follows) =====
add(ok): sub-02/onsets/run-1/body.txt (file)
add(ok): sub-02/onsets/run-1/face.txt (file)
add(ok): sub-02/onsets/run-1/house.txt (file)
add(ok): sub-02/onsets/run-1/object.txt (file)
add(ok): sub-02/onsets/run-1/scene.txt (file)
add(ok): sub-02/onsets/run-1/scramble.txt (file)
save(ok): . (dataset)
```

The dataset now contains and manages all of the required inputs, and we're ready for FSL. Since FSL is not a simple program, we make sure to record the precise software environment for the analysis with **datalad containers-run**. First, we get a container with FSL in the version we require:

```
$ datalad containers-add fsl --url shub://mih/ohbm2018-training:fsl
[INFO] Initiating special remote datalad
add(ok): .datalad/config (file)
save(ok): . (dataset)
containers_add(ok): /home/me/usecases/repro2/glm_analysis/.datalad/environments/fsl/image␣
↪(file)
action summary:
  add (ok: 1)
  containers_add (ok: 1)
  save (ok: 1)
```

As the analysis setup is now complete, let's label this state of the dataset:

```
$ datalad save --version-tag ready4analysis
save(ok): . (dataset)
```

All we have left is to configure the desired first-level GLM analysis with FSL. At this point, the template contains placeholders for the basepath and the subject ID, and they need to be replaced. The following command uses the arcane, yet powerful SED editor to do this. We will again use **datalad run** to invoke our command so that we store in the history how this template was generated (so that we may audit, alter, or regenerate this file in the future — fearlessly).

```
$ datalad run \
 -m "FSL FEAT analysis config script" \
 --output sub-02/1stlvl_design.fsf \
 bash -c 'sed -e "s,##BASEPATH##,{pwd},g" -e "s,##SUB##,sub-02,g" \
 code/ffa_design.fsf > {outputs}'
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): sub-02/1stlvl_design.fsf (file)
save(ok): . (dataset)
```

To compute the analysis, a simple `feat sub-02/1stlvl_design.fsf` command is wrapped into a datalad containers-run command with appropriate --input and --output specification:

```
$ datalad containers-run --container-name fsl -m "sub-02 1st-level GLM" \
  --input sub-02/1stlvl_design.fsf \
  --input sub-02/onsets \
  --input inputs/rawdata/sub-02/func/sub-02_task-oneback_run-01_bold.nii.gz \
  --output sub-02/1stlvl_glm.feat \
  feat {inputs[0]}
[INFO] Making sure inputs are available (this may take some time)
get(ok): inputs/rawdata/sub-02/func/sub-02_task-oneback_run-01_bold.nii.gz (file) [from␣
↪origin...]
[INFO] == Command start (output follows) =====
To view the FEAT progress and final report, point your web browser at /home/me/usecases/
↪repro2/glm_analysis/sub-02/1stlvl_glm.feat/report_log.html
[INFO] == Command exit (modification check follows) =====
add(ok): sub-02/1stlvl_glm.feat/.files/fsl.css (file)
save(ok): . (dataset)
action summary:
```

```
add (ok: 344)
get (notneeded: 4, ok: 1)
save (notneeded: 1, ok: 1)
```

Once this command finishes, DataLad will have captured the entire FSL output, and the dataset
will contain a complete record all the way from the input BIDS dataset to the GLM results. The
BIDS subdataset in turn has a complete record of all processing down from the raw DICOMs
onwards.

> ⚠️ **Many files need more planning**
>
> See how many files were created and added in this computation of a single participant? If
> your study has many participants, analyses like the one above could inflate your dataset.
> Please check out the chapter *Go big or go home* (page 323). in particular the section
> *Calculate in greater numbers* (page 326) for tips and tricks on how to create analyses
> datasets that scale.

Archive data and results

After study completion it is important to properly archive data and results, for example for
future inquiries by reviewers or readers of the associated publication. Thanks to the modularity
of the study units, this tasks is easy and avoids needless duplication.

The raw data is tracked in its own dataset (`localizer_scans`) that only needs to be archived
once, regardless of how many analysis are using it as input. This means that we can "throw
away" this subdataset copy within this analysis dataset. DataLad can re-obtain the correct
version at any point in the future, as long as the recorded location remains accessible.

To make sure we're not deleting information we are not aware of, **datalad diff** and **git log**
can help to verify that the subdataset is in the same state as when it was initially added:

```
$ datalad diff -- inputs
```

The command does not show any output, thus indicating that there is indeed no difference. `git
log` confirms that the only action that was performed on `inputs/` was the addition of it as a
subdataset:

```
$ git log -- inputs
commit 05703a378ad34120fcb5cf6556446b78da8a98a8
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jul 29 10:15:40 2021 +0200

    [DATALAD] Added subdataset
```

Since the state of the subdataset is exactly the state of the original `localizer_scans` dataset it
is safe to uninstall it.

```
$ datalad uninstall --dataset . inputs --recursive
drop(ok): /home/me/usecases/repro2/glm_analysis/inputs/rawdata/sub-02/func/sub-02_task-
↪oneback_run-01_bold.nii.gz (file)
drop(ok): /home/me/usecases/repro2/glm_analysis/inputs/rawdata/sub-02/func/sub-02_task-
↪oneback_run-01_events.tsv (file)
```

```
drop(ok): inputs/rawdata (directory)
uninstall(ok): inputs/rawdata (dataset)
action summary:
  drop (ok: 3)
  uninstall (ok: 1)
```

Prior to archiving the results, we can go one step further and verify their computational reproducibility. DataLad's rerun command is capable of "replaying" any recorded command. The following command re-executes the FSL analysis by re-running everything since the dataset was tagged as ready4analysis). It will record the recomputed results in a separate Git branch named verify. Afterwards, we can automatically compare these new results to the original ones in the master branch. We will see that all outputs can be reproduced in bit-identical form. The only changes are observed in log files that contain volatile information, such as time steps.

```
$ datalad rerun --branch verify --onto ready4analysis --since ready4analysis
[INFO] checkout commit 6bd6c90;
[INFO] run commit f10c6c1; (FSL FEAT analysis...)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): sub-02/1stlvl_design.fsf (file)
save(ok): . (dataset)
[INFO] run commit 0238dbf; (sub-02 1st-level GLM)
[INFO] Making sure inputs are available (this may take some time)
[INFO] Cloning dataset to Dataset(/home/me/usecases/repro2/glm_analysis/inputs/rawdata)
[INFO] Attempting to clone from ../localizer_scans to /home/me/usecases/repro2/glm_
↪analysis/inputs/rawdata
[INFO] Completed clone attempts for Dataset(/home/me/usecases/repro2/glm_analysis/inputs/
↪rawdata)
[INFO] scanning for unlocked files (this may take some time)
get(ok): inputs/rawdata/sub-02/func/sub-02_task-oneback_run-01_bold.nii.gz (file) [from
↪origin...]
[INFO] == Command start (output follows) =====
To view the FEAT progress and final report, point your web browser at /home/me/usecases/
↪repro2/glm_analysis/sub-02/1stlvl_glm.feat/report_log.html
[INFO] == Command exit (modification check follows) =====
save(ok): . (dataset)
action summary:
  add (ok: 345)
  get (notneeded: 4, ok: 1)
  save (notneeded: 1, ok: 2)


# check that we are now on the new `verify` branch
$ git branch
  git-annex
  master
* verify


# compare which files have changes with respect to the original results
$ git diff master --stat
 sub-02/1stlvl_glm.feat/logs/feat0                               | 2 +-
 .../1stlvl_glm.feat/logs/{feat0_init.e3076726 => feat0_init.e3079448}  | 0
 .../1stlvl_glm.feat/logs/{feat0_init.o3076726 => feat0_init.o3079448}  | 0
 sub-02/1stlvl_glm.feat/logs/feat1                               | 2 +-
 sub-02/1stlvl_glm.feat/logs/{feat2_pre.e3076809 => feat2_pre.e3079531} | 0
```

```
sub-02/1stlvl_glm.feat/logs/{feat2_pre.o3076809 => feat2_pre.o3079531}  | 0
.../1stlvl_glm.feat/logs/{feat3_film.e3077273 => feat3_film.e3080001}    | 0
.../1stlvl_glm.feat/logs/{feat3_film.o3077273 => feat3_film.o3080001}    | 0
.../1stlvl_glm.feat/logs/{feat4_post.e3077646 => feat4_post.e3080379}    | 0
.../1stlvl_glm.feat/logs/{feat4_post.o3077646 => feat4_post.o3080379}    | 0
.../1stlvl_glm.feat/logs/{feat5_stop.e3078278 => feat5_stop.e3081019}    | 0
.../1stlvl_glm.feat/logs/{feat5_stop.o3078278 => feat5_stop.o3081019}    | 0
sub-02/1stlvl_glm.feat/report.html                                       | 2 +-
sub-02/1stlvl_glm.feat/report_log.html                                   | 2 +-
14 files changed, 4 insertions(+), 4 deletions(-)


# switch back to the master branch and remove the `verify` branch
$ git checkout master
$ git branch -D verify
Switched to branch 'master'
Deleted branch verify (was 8b6a3a7).
```

The outcome of this usecase can be found as a dataset on Github here[600].

---

[600] https://github.com/myyoda/demo-dataset-glmanalysis

SCALING UP: MANAGING 80TB AND 15 MILLION FILES FROM THE HCP RELEASE

This usecase outlines how a large data collection can be version controlled and published in an accessible manner with DataLad in a remote indexed archive (RIA) data store. Using the Human Connectome Project[603] (HCP) data as an example, it shows how large-scale datasets can be managed with the help of modular nesting, and how access to data that is contingent on usage agreements and external service credentials is possible via DataLad without circumventing or breaching the data providers terms:

1. The **datalad addurls** command is used to automatically aggregate files and information about their sources from public AWS S3[604] bucket storage into small-sized, modular Data-Lad datasets.

2. Modular datasets are structured into a hierarchy of nested datasets, with a single HCP superdataset at the top. This modularizes storage and access, and mitigates performance problems that would arise in oversized standalone datasets, but maintains access to any subdataset from the top-level dataset.

3. Individual datasets are stored in a remote indexed archive (RIA) store at store.datalad.org[605] under their DATASET ID. This setup constitutes a flexible, domain-agnostic, and scalable storage solution, while dataset configurations enable seamless automatic dataset retrieval from the store.

4. The top-level dataset is published to GitHub as a public access point for the full HCP dataset. As the RIA store contains datasets with only file source information instead of hosting data contents, a **datalad get** retrieves file contents from the original AWS S3 sources.

5. With DataLad's authentication management, users will authenticate once – and are thus required to accept the HCP projects terms to obtain valid credentials –, but subsequent **datalad get** commands work swiftly without logging in.

6. The **datalad copy-file** can be used to subsample special-purpose datasets for faster access.

---

[603] http://www.humanconnectomeproject.org/
[604] https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html
[605] http://store.datalad.org/

27.1 The Challenge

The Human Connectome Project[606] aims to provide an unparalleled compilation of neural data through a customized database. Its largest open access data collection is the WU-Minn HCP1200 Data[607]. It is made available via a public AWS S3 bucket and includes high-resolution 3T magnetic resonance[608] scans from young healthy adult twins and non-twin siblings (ages 22-35) using four imaging modalities: structural images (T1w and T2w), resting-state fMRI (rfMRI)[609], task-fMRI (tfMRI), and high angular resolution diffusion imaging (dMRI)[610]. It further includes behavioral and other individual subject measure data for all, and magnetoencephalography[611] data and 7T MR data for a subset of subjects (twin pairs). In total, the data release encompasses around 80TB of data in 15 million files, and is of immense value to the field of neuroscience.

Its large amount of data, however, also constitutes a data management challenge: Such amounts of data are difficult to store, structure, access, and version control. Even tools such as DataLad, and its foundations, GIT and GIT-ANNEX, will struggle or fail with datasets of this size or number of files. Simply transforming the complete data release into a single DataLad dataset would at best lead to severe performance issues, but quite likely result in software errors and crashes. Moreover, access to the HCP data is contingent on consent to the data usage agreement[612] of the HCP project and requires valid AWS S3 credentials. Instead of hosting this data or providing otherwise unrestrained access to it, an HCP DataLad dataset would need to enable data retrieval from the original sources, conditional on the user agreeing to the HCP usage terms.

27.2 The DataLad Approach

Using the `datalad addurls` command, the HCP data release is aggregated into a large amount (N $\sim$= 4500) of datasets. A lean top-level dataset combines all datasets into a nested dataset hierarchy that recreates the original HCP data release's structure. The topmost dataset contains one subdataset per subject with the subject's release notes, and within each subject's subdataset, each additional available subdirectory is another subdataset. This preserves the original structure of the HCP data release, but builds it up from sensible components that resemble standalone dataset units. As with any DataLad dataset, dataset nesting and operations across dataset boundaries are seamless, and allow to easily retrieve data on a subject, modality, or file level.

The highly modular structure has several advantages. For one, with barely any data in the superdataset, the top-level dataset is very lean. It mainly consists of an impressive `.gitmodules` file[624] with almost 1200 registered (subject-level) subdatasets. The superdataset is published to GITHUB at github.com/datalad-datasets/human-connectome-project-openaccess[613] to expose this superdataset and allow anyone to install it with a single `datalad clone` command in a few seconds. Secondly, the modularity from splitting the data release into several thousand subdatasets has performance advantages. If GIT or GIT-ANNEX repositories exceed a certain size

---

[606] http://www.humanconnectomeproject.org/

[607] https://humanconnectome.org/study/hcp-young-adult/document/1200-subjects-data-release/

[608] https://en.wikipedia.org/wiki/Magnetic_resonance_imaging

[609] https://en.wikipedia.org/wiki/Resting_state_fMRI

[610] https://en.wikipedia.org/wiki/Diffusion_MRI

[611] https://en.wikipedia.org/wiki/Magnetoencephalography

[612] http://www.humanconnectomeproject.org/wp-content/uploads/2010/01/HCP_Data_Agreement.pdf

[624] If you want to read up on how DataLad stores information about registered subdatasets in `.gitmodules`, checkout section *More on DIY configurations* (page 117).

[613] https://github.com/datalad-datasets/human-connectome-project-openaccess

(either in terms of file sizes or the number of files), performance can drop severely[625]. By dividing the vast amount of data into many subdatasets, this can be prevented: Subdatasets are small-sized units that are combined to the complete HCP dataset structure, and nesting comes with no additional costs or difficulties, as DataLad can work smoothly across hierarchies of subdatasets.

In order to simplify access to the data instead of providing data access that could circumvent HCP license term agreements for users, DataLad does not host any HCP data. Instead, thanks to **datalad addurls**, each data file knows its source (the public AWS S3 bucket of the HCP project), and a **datalad get** will retrieve HCP data from this bucket. With this setup, anyone who wants to obtain the data will still need to consent to data usage terms and retrieve AWS credentials from the HCP project, but can afterwards obtain the data solely with DataLad commands from the command line or in scripts. Only the first **datalad get** requires authentication with AWS credentials provided by the HCP project: DataLad will prompt any user at the time of retrieval of the first file content of the dataset. Afterwards, no further authentication is needed, unless the credentials become invalid or need to be updated for other reasons. Thus, in order to retrieve HCP data of up to single file level with DataLad, users only need to:

- **datalad clone** the superdataset from GITHUB (github.com/datalad-datasets/human-connectome-project-openaccess[614])

- Create an account at http://db.humanconnectome.org to accept data use terms and obtain AWS credentials

- Use **datalad get [-n] [-r] PATH** to retrieve file, directory, or subdataset contents on demand. Authentication is necessary only once (at the time of the first **datalad get**).

The HCP data release, despite its large size, can thus be version controlled and easily distributed with DataLad. In order to speed up data retrieval, subdataset installation can be parallelized, and the full HCP dataset can be subsampled into special-purpose datasets using DataLad's **copy-file** command (introduced with DataLad version 0.13.0)

27.3 Step-by-Step

Building and publishing a DataLad dataset with HCP data consists of several steps: 1) Creating all necessary datasets, 2) publishing them to a RIA store, and 3) creating an access point to all files in the HCP data release. The upcoming subsections detail each of these.

Dataset creation with `datalad addurls`

The **datalad addurls** command (datalad-addurls manual) allows you to create (and update) potentially nested DataLad datasets from a list of download URLs that point to the HCP files in the S3 buckets. By supplying subject specific .csv files that contain S3 download links, a subject ID, a file name, and a version specification per file in the HCP dataset, as well as information on where subdataset boundaries are, **datalad addurls** can download all subjects' files and create (nested) datasets to store them in. With the help of a few bash commands, this task can be automated, and with the help of a job scheduler[615], it can also be parallelized. As soon as files

---

[625] Precise performance will always be dependent on the details of the repository, software setup, and hardware, but to get a feeling for the possible performance issues in oversized datasets, imagine a mere **git status** or **datalad status** command taking several minutes up to hours in a clean dataset.

[614] https://github.com/datalad-datasets/human-connectome-project-openaccess

[615] https://en.wikipedia.org/wiki/Job_scheduler

are downloaded and saved to a dataset, their content can be dropped with **datalad drop**: The origin of the file was successfully recorded, and a **datalad get** can now retrieve file contents on demand. Thus, shortly after a complete download of the HCP project data, the datasets in which it has been aggregated are small in size, and yet provide access to the HCP data for anyone who has valid AWS S3 credentials.

At the end of this step, there is one nested dataset per subject in the HCP data release. If you are interested in the details of this process, checkout the hidden section below.

> **M27.1 How exactly did the datasets came to be?**
>
> All code and tables necessary to generate the HCP datasets can be found on GitHub at github.com/TobiasKadelka/build_hcp[616].
>
> The **datalad addurls** command is capable of building all necessary nested subject datasets automatically, it only needs an appropriate specification of its tasks. We'll approach the function of **datalad addurls** and how exactly it was invoked to build the HCP dataset by looking at the information it needs. Below are excerpts of the .csv table of one subject (100206) that illustrate how **addurls** works:
>
> **Listing 1:** Table header and some of the release note files
>
> ```
> "original_url","subject","filename","version"
> "s3://hcp-openaccess/HCP_1200/100206/release-notes/Diffusion_unproc.txt","100206",
> ↪"release-notes/Diffusion_unproc.txt","j9bm9Jvph3EzC0t9Jl51KVrq6NFuoznu"
> "s3://hcp-openaccess/HCP_1200/100206/release-notes/ReleaseNotes.txt","100206",
> ↪"release-notes/ReleaseNotes.txt","RgG.VC2mzp5xIc6ZGN6vB7iZ0mG7peXN"
> "s3://hcp-openaccess/HCP_1200/100206/release-notes/Structural_preproc.txt","100206",
> ↪"release-notes/Structural_preproc.txt","OeUYjysiX5zR7nRMixCimFa_6yQ3IKqf"
> "s3://hcp-openaccess/HCP_1200/100206/release-notes/Structural_preproc_extended.txt",
> ↪"100206","release-notes/Structural_preproc_extended.txt","cyP8G5_
> ↪YX5F30gO9Yrpk8TADhkLltrNV"
> "s3://hcp-openaccess/HCP_1200/100206/release-notes/Structural_unproc.txt","100206",
> ↪"release-notes/Structural_unproc.txt","AyW6GmavML6I7LfbULVmtGIwRGpFmfPZ"
> ```

💡

Listing 2: Some files in the MNINonLinear directory

```
"s3://hcp-openaccess/HCP_1200/100206/MNINonLinear/100206.164k_fs_LR.wb.spec","100206
↪","MNINonLinear//100206.164k_fs_LR.wb.spec","JSZJhZekZnMhv1sDWih.khEVUNZXMHTE"
"s3://hcp-openaccess/HCP_1200/100206/MNINonLinear/100206.ArealDistortion_FS.164k_fs_
↪LR.dscalar.nii","100206","MNINonLinear//100206.ArealDistortion_FS.164k_fs_LR.
↪dscalar.nii","sP4uw8R1oJyqCWeInSd9jmOBjfOCtN4D"
"s3://hcp-openaccess/HCP_1200/100206/MNINonLinear/100206.ArealDistortion_MSMAll.
↪164k_fs_LR.dscalar.nii","100206","MNINonLinear//100206.ArealDistortion_MSMAll.
↪164k_fs_LR.dscalar.nii","yD88c.HfsFwjyNXHQQv2SymGIsSYHQVZ"
"s3://hcp-openaccess/HCP_1200/100206/MNINonLinear/100206.ArealDistortion_MSMSulc.
↪164k_fs_LR.dscalar.nii","100206","MNINonLinear
```

The `.csv` table contains one row per file, and includes the columns `original_url`, `subject`, `filename`, and `version`. `original_url` is an s3 URL pointing to an individual file in the S3 bucket, `subject` is the subject's ID (here: `100206`), `filename` is the path to the file within the dataset that will be build, and `version` is an S3 specific file version identifier. The first table excerpt thus specifies a few files in the directory `release-notes` in the dataset of subject `100206`. For **datalad addurls**, the column headers serve as placeholders for fields in each row. If this table excerpt is given to a **datalad addurls** call as shown below, it will create a dataset and download and save the precise version of each file in it:

```
$ datalad addurls -d <Subject-ID> <TABLE> '{original_url}?versionId={version}' '
↪{filename}'
```

This command translates to "create a dataset with the name of the subject ID (`-d <Subject-ID>`) and use the provided table (`<TABLE>`) to assemble the dataset contents. Iterate through the table rows, and perform one download per row. Generate the download URL from the `original_url` and `version` field of the table (`{original_url}?versionId={version}'`), and save the downloaded file under the name specified in the `filename` field (`'{filename}')`".

If the file name contains a double slash (`//`), for example seen in the second table excerpt in `"MNINonLinear//...`, this file will be created underneath a *subdataset* of the name in front of the double slash. The rows in the second table thus translate to "save these files into the subdataset `MNINonLinear`, and if this subdataset does not exist, create it".

Thus, with a single subject's table, a nested, subject specific dataset is built. Here is how the directory hierarchy looks for this particular subject once **datalad addurls** worked through its table:

```
100206
├── MNINonLinear    <- subdataset
├── release-notes
├── T1w             <- subdataset
└── unprocessed     <- subdataset
```

This is all there is to assemble subject specific datasets. The interesting question is: How can this be done as automated as possible?

How to create subject-specific tables

One crucial part of the process are the subject specific tables for **datalad addurls**. The information on the file url, its name, and its version can be queried with the **datalad ls** command (datalad-ls manual). It is a DataLad-specific version of the Unix `ls` command and can be used to list summary information about s3 URLs and datasets. With this command, the public S3 bucket can be queried and the command will output the relevant

information.

The `datalad ls` command is a rather old command and less user-friendly than other commands demonstrated in the handbook. One problem for automation is that the command is made for interactive use, and it outputs information in a non-structured fashion. In order to retrieve the relevant information, a custom Python script was used to split its output and extract it. This script can be found in the GitHub repository as code/create_subject_table.py[617].

How to schedule datalad addurls commands for all tables

Once the subject specific tables exist, `datalad addurls` can start to aggregate the files into datasets. To do it efficiently, this can be done in parallel by using a job scheduler. On the computer cluster the datasets were aggregated, this was HTCondor[618].

The jobs (per subject) performed by HTCondor consisted of

- a `datalad addurls` command to generate the (nested) dataset and retrieve content once[626]:

```
datalad -l warning addurls -d "$outds" -c hcp_dataset "$subj_table" '
↪{original_url}?versionId={version}' '{filename}'
```

- a subsequent `datalad drop` command to remove file contents as soon as they were saved to the dataset to save disk space (this is possible since the S3 source of the file is known, and content can be reobtained using `get`):

```
datalad drop -d "$outds" -r --nocheck
```

- a few (Git) commands to clean up well afterwards, as the system the HCP dataset was downloaded to had a strict 5TB limit on disk usage.

Summary

Thus, in order to download the complete HCP project and aggregate it into nested subject level datasets (on a system with much less disk space than the complete HCP project's size!), only two DataLad commands, one custom configuration, and some scripts to parse terminal output into `.csv` tables and create subject-wise HTCondor jobs were necessary. With all tables set up, the jobs ran over the Christmas break and finished before everyone went back to work. Getting 15 million files into datasets? Check!

---

[617] https://github.com/TobiasKadelka/build_hcp/blob/master/code/create_subject_table.py

[618] https://research.cs.wisc.edu/htcondor/

[626] Note that this command is more complex than the previously shown `datalad addurls` command. In particular, it has an additional *loglevel* configuration for the main command, and creates the datasets with an *hcp_dataset* configuration. The logging level was set (to warning) to help with post-execution diagnostics in the HTCondors log files. The configuration can be found in code/cfg_hcp_dataset[627] and enables a SPECIAL REMOTE in the resulting dataset.

Using a Remote Indexed Archive Store for dataset hosting

All datasets were built on a scientific compute cluster. In this location, however, datasets would only be accessible to users with an account on this system. Subsequently, therefore, everything was published with `datalad push` to the publicly available store.datalad.org[619], a remote indexed archive (RIA) store.

A RIA store is a flexible and scalable data storage solution for DataLad datasets. While its layout may look confusing if one were to take a look at it, a RIA store is nothing but a clever storage solution, and users never consciously interact with the store to get the HCP datasets. On the

---

[619] http://store.datalad.org/

lowest level, store.datalad.org[620] is a directory on a publicly accessible server that holds a great number of datasets stored as BARE GIT REPOSITORIES. The only important aspect of it for this usecase is that instead of by their names (e.g., 100206), datasets are stored and identified via their DATASET ID. The **datalad clone** command can understand this layout and install datasets from a RIA store based on their ID.

> **💡 M27.2 How would a datalad clone from a RIA store look like?**
>
> In order to get a dataset from a RIA store, **datalad clone** needs a RIA URL. It is build from the following components:
> - a ria+ identifier
> - a path/url to the store in question. For store.datalad.org, this is http://store.datalad.org, but it could also be an SSH url, such as ssh://juseless.inm7.de/data/group/psyinf/dataset_store
> - a pound sign (#)
> - the dataset ID
> - and optionally a version or branch specification (appended with a leading @)
>
> Here is how a valid **datalad clone** command from the data store for one dataset would look like:
>
> ```
> datalad clone 'ria+http://store.datalad.org#d1ca308e-3d17-11ea-bf3b-f0d5bf7b5561'␣
> ↪subj-01
> ```
>
> But worry not! To get the HCP data, no-one will ever need to compose **clone** commands to RIA stores apart from DataLad itself.

A RIA store is used, because – among other advantages – its layout makes the store flexible and scalable. With datasets of sizes like the HCP project, especially scalability becomes an important factor. If you are interested in finding out why, you can find more technical details on RIA stores, their advantages, and even how to create and use one yourself in the section *Remote Indexed Archives for dataset storage and backup* (page 294).

Making the datasets accessible

At this point, roughly 1200 nested datasets were created and published to a publicly accessible RIA store. This modularized the HCP dataset and prevented performance issues that would arise in oversized datasets. In order to make the complete dataset available and accessible from one central point, the only thing missing is a single superdataset.

For this, a new dataset, human-connectome-project-openaccess, was created. It contains a README file with short instructions on how to use it, a text-based copy of the HCP project's data usage agreement, – and each subject dataset as a subdataset. The .gitmodules file[624] of this superdataset thus is impressive. Here is an excerpt:

```
[submodule "100206"]
    path = HCP1200/100206
    url = ./HCP1200/100206
    branch = master
    datalad-id = 346a3ae0-2c2e-11ea-a27d-002590496000
[submodule "100307"]
    path = HCP1200/100307
```

(continues on next page)

---

[620] http://store.datalad.org/

```
    url = ./HCP1200/100307
    branch = master
    datalad-id = a51b84fc-2c2d-11ea-9359-0025904abcb0
[submodule "100408"]
    path = HCP1200/100408
    url = ./HCP1200/100408
    branch = master
    datalad-id = d3fa72e4-2c2b-11ea-948f-0025904abcb0
[...]
```

For each subdataset (named after subject IDs), there is one entry (note that individual `url`s of the subdatasets are pointless and not needed: As will be demonstrated shortly, DataLad resolves each subdataset ID from the common store automatically). Thus, this superdataset combines all individual datasets to the original HCP dataset structure. This (and only this) superdataset is published to a public GITHUB repository that anyone can `datalad clone`[628].

Data retrieval and interacting with the repository

> ⚠️ **HCP dataset version requirements**
>
> Using this dataset requires DataLad version 0.12.2 or higher. Upgrading an existing DataLad installation is detailed in section *Installation and configuration* (page 10).

Procedurally, getting data from this dataset is almost as simple as with any other public DataLad dataset: One needs to `clone` the repository and use `datalad get [-n] [-r] PATH` to retrieve any file, directory, or subdataset (content). But because the data will be downloaded from the HCP's AWS S3 bucket, users will need to create an account at db.humanconnectome.org[621] to agree to the project's data usage terms and get credentials. When performing the first `datalad get` for file contents, DataLad will prompt for these credentials interactively from the terminal. Once supplied, all subsequent `get` commands will retrieve data right away.

> 💡 **M27.3 Resetting AWS credentials**
>
> In case one misenters their AWS credentials or needs to reset them, this can easily be done using the Python keyring[622] package. For more information on `keyring` and DataLad's authentication process, see the *Basic process* section in *Configure custom data access* (page 290).
>
> After launching Python, import the `keyring` package and use the `set_password()` function. This function takes 3 arguments:
> - `system`: "datalad-hcp-s3" in this case
> - `username`: "key_id" if modifying the AWS access key ID or "secret_id" if modifying the secret access key
> - `password`: the access key itself

---

[628] To re-read about publishing datasets to hosting services such as GITHUB or GITLAB, go back to *Publishing the dataset to GitHub* (page 153).

[621] http://db.humanconnectome.org

```
import keyring

keyring.set_password("datalad-hcp-s3", "key_id", <password>)
keyring.set_password("datalad-hcp-s3", "secret_id", <password>)
```

Alternatively, one can set their credentials using environment variables. For more details on this method, *see this Findoutmore* (page 125).

```
$ export DATALAD_hcp_s3_key_id=<password>
$ export DATALAD_hcp_s3_secret_id=<password>
```

---

[622] https://keyring.readthedocs.io/en/latest/

Internally, DataLad cleverly manages the crucial aspects of data retrieval: Linking registered subdatasets to the correct dataset in the RIA store. If you inspect the GitHub repository, you will find that the subdataset links in it will not resolve if you click on them, because none of the subdatasets were published to GitHub[629], but lie in the RIA store instead. Dataset or file content retrieval will nevertheless work automatically with **datalad get**: Each .gitmodule entry lists the subdataset's dataset ID. Based on a configuration of "subdataset-source-candidates" in .datalad/config of the superdataset, the subdataset ID is assembled to a RIA URL that retrieves the correct dataset from the store by **get**:

```
$ cat .datalad/config
[datalad "dataset"]
    id = 2e2a8a70-3eaa-11ea-a9a5-b4969157768c
[datalad "get"]
    subdataset-source-candidate-origin = "ria+http://store.datalad.org#{id}"
```

This configuration allows **get** to flexibly generate RIA URLs from the base URL in the config file and the dataset IDs listed in .gitmodules. In the superdataset, it needed to be done "by hand" via the **git config** command. Because the configuration should be shared together with the dataset, the configuration needed to be set in .datalad/config[630]:

```
$ git config -f .datalad/config "datalad.get.subdataset-source-candidate-origin"
↪"ria+http://store.datalad.org#{id}"
```

With this configuration, **get** will retrieve all subdatasets from the RIA store. Any subdataset that is obtained from a RIA store in turn gets the very same configuration automatically into .git/config. Thus, the configuration that makes seamless subdataset retrieval from RIA stores possible is propagated throughout the dataset hierarchy. With this in place, anyone can clone the top most dataset from GitHub, and – given they have valid credentials – get any file in the HCP dataset hierarchy.

---

[629] If you coded along in the Basics part of the book and published your dataset to GIN, you have experienced in *Subdataset publishing* (page 212) how the links to unpublished subdatasets in a published dataset do not resolve in the webinterface: Its path points to a URL that would resolve to lying underneath the superdataset, but there is not published subdataset on the hosting platform!

[630] To re-read on configurations of datasets, go back to sections *DIY configurations* (page 112) and *More on DIY configurations* (page 117).

Speeding operations up

At this point in time, the HCP dataset is a single, published superdataset with ~4500 subdatasets that are hosted in a REMOTE INDEXED ARCHIVE (RIA) STORE at store.datalad.org[623]. This makes the HCP data accessible via DataLad and its download easier. One downside to gigantic nested datasets like this one, though, is the time it takes to retrieve all of it. Some tricks can help to mitigate this: Contents can either be retrieved in parallel, or, in the case of general need for subsets of the dataset, subsampled datasets can be created with `datalad copy-file`.

If the complete HCP dataset is required, subdataset installation and data retrieval can be sped up by parallelizing. The gists *Parallelize subdataset processing* (page 273) and *Retrieve partial content from a hierarchy of (uninstalled) datasets* (page 276) can shed some light on how to do this. If you are interested in learning about the `datalad copy-file`, checkout the section *Subsample datasets using datalad copy-file* (page 313).

Summary

This usecase demonstrated how it is possible to version control and distribute datasets of sizes that would otherwise be unmanageably large for version control systems. With the public HCP dataset available as a DataLad dataset, data access is simplified, data analysis that use the HCP data can link it (in precise versions) to their scripts and even share it, and the complete HCP release can be stored at a fraction of its total size for on demand retrieval.

---

[623] http://store.datalad.org/

BUILDING A SCALABLE DATA STORAGE FOR SCIENTIFIC COMPUTING

Research can require enormous amounts of data. Such data needs to be accessed by multiple people at the same time, and is used across a diverse range of computations or research questions. The size of the dataset, the need for simultaneous access and transformation of this data by multiple people, and the subsequent storing of multiple copies or derivatives of the data constitutes a challenge for computational clusters and requires state-of-the-art data management solutions. This use case details a model implementation for a scalable data storage solution, suitable to serve the computational and logistic demands of data science in big (scientific) institutions, while keeping workflows for users as simple as possible. It elaborates on

1. How to implement a scalable REMOTE INDEXED ARCHIVE (RIA) STORE to flexibly store large amounts of DataLad datasets, potentially remote to lower storage strains on computing infrastructure,

2. How disk-space aware computing can be eased by DataLad based workflows and enforced by infrastructural incentives and limitations, and

3. How to reduce technical complexities for users and encourage reproducible, version-controlled, and scalable scientific workflows.

> ⚠️ **Use case target audience**
>
> This usecase is technical in nature and aimed at IT/data management personnel seeking insights into the technical implementation and configuration of a RIA store or into its workflows. In particular, it describes the RIA data storage and workflow implementation as done in INM-7, research centre Juelich, Germany.
> **Note further**: Building a RIA store requires **DataLad version 0.13.0** or higher.

28.1 The Challenge

The data science institute XYZ consists of dozens of people: Principle investigators, PhD students, general research staff, system administration, and IT support. It does research on important global issues, and prides itself with ground-breaking insights obtained from elaborate and complex computations run on a large scientific computing cluster. The datasets used in the institute are big both in size and number of files, and expensive to collect. Therefore, datasets are used for various different research questions, by multiple researchers. Every member of the institute has an account on an expensive and large compute cluster, and all of the data exists in dedicated directories on this server. However, researchers struggle with the technical overhead of data management *and* data science. In order to work on their research questions without

modifying original data, every user creates their own copies of the full data in their user account on the cluster – even if it contains many files that are not necessary for their analysis. In addition, as version control is not a standard skill, they add all computed derivatives and outputs, even old versions, out of fear of losing work that may become relevant again. Thus, an excess of (unorganized) data copies and derivatives exists in addition to the already substantial amount of original data. At the same time, the compute cluster is both the data storage and the analysis playground for the institute. With data directories of several TB in size, *and* computationally heavy analyses, the compute cluster is quickly brought to its knees: Insufficient memory and IOPS starvation make computations painstakingly slow, and hinder scientific progress. Despite the elaborate and expensive cluster setup, exciting datasets can not be stored or processed, as there just doesn't seem to be enough disk space.

Therefore, the challenge is two-fold: On an infrastructural level, institute XYZ needs a scalable, flexible, and maintainable data storage solution for their growing collection of large datasets. On the level of human behavior, researchers not formerly trained in data management need to apply and adhere to advanced data management principles.

28.2 The DataLad approach

The compute cluster is refurbished to a state-of-the-art data management system. For a scalable and flexible dataset storage, the data store is a REMOTE INDEXED ARCHIVE (RIA) STORE – an extendable, file-system based storage solution for DataLad datasets that aligns well with the requirements of scientific computing (infrastructure). The RIA store is configured as a git-annex ORA-remote ("optional remote archive") special remote for access to annexed keys in the store and so that full datasets can be (compressed) 7-zip archives. The latter is especially useful in case of filesystem inode limitations, such as on HPC storage systems: Regardless of a dataset's number of files and size, (compressed) 7zipped datasets use only few inodes, but retain the ability to query available files. Unlike traditional solutions, both because of the size of the large amounts of data, and for more efficient use of compute power for calculations instead of data storage, the RIA store is set up *remote*: Data is stored on a different machine than the one the scientific analyses are computed on. While unconventional, it is convenient, and perfectly possible with DataLad.

The infrastructural changes are accompanied by changes in the mindset and workflows of the researchers that perform analyses on the cluster. By using a RIA store, the institute's work routines are adjusted around DataLad datasets. Simple configurations, distributed system-wide with DataLad's run-procedures, or basic data management principles improve the efficiency and reproducibility of research projects: Analyses are set-up inside of DataLad datasets, and for every analysis, an associated `project` is created under the namespace of the institute on the institute's GITLAB instance automatically. This does not only lead to vastly simplified version control workflows, but also to simplified access to projects and research logs for collaborators and supervisors. Input data gets installed as subdatasets from the RIA store. This automatically links analyses projects to data sets, and allows for fine-grained access of up to individual file level. With only precisely needed data, analyses datasets are already much leaner than with previous complete dataset copies, but as data can be re-obtained on-demand from the store, original input files or files that are easily recomputed can safely be dropped to save even more disk-space. Beyond this, upon creation of an analysis project, the associated GitLab project is automatically configured as a remote with a publication dependency on the data store, thus enabling vastly simplified data publication routines and backups of pristine results: After computing their results, a `datalad push` is all it takes to backup and share ones scientific insights. Thus, even with a complex setup of data store, compute infrastructure, and repository hosting,

configurations adjusted to the compute infrastructure can be distributed and used to mitigate any potential remaining technical overhead. Finally, with all datasets stored in a RIA store and in a single place, any remaining maintenance and query tasks in the datasets can be performed by data management personnel without requiring domain knowledge about dataset contents.

28.3 Step-by-step

The following section will elaborate on the details of the technical implementation of a RIA store, and the workflow requirements and incentives for researchers. Both of them are aimed at making scientific analyses on a compute cluster scale and can be viewed as complimentary but independent.

> ⚠️ **Note on the generality of the described setup**
>
> Some hardware-specific implementation details are unique to the real-world example this usecase is based on, and are not a requirement. In this particular case of application, for example, a *remote* setup for a RIA store made sense: Parts of an old compute cluster and of the super computer at the Juelich supercomputing centre (JSC) instead of the institutes compute cluster are used to host the data store. This may be an unconventional storage location, but it is convenient: The data does not strain the compute cluster, and with DataLad, it is irrelevant where the RIA store is located. The next subsection introduces the general layout of the compute infrastructure and some DataLad-unrelated incentives and restrictions.

Incentives and imperatives for disk-space aware computing

On a high level, the layout and relationships of the relevant computational infrastructure in this usecase are as follows: Every researcher has a workstation that they can access the compute cluster with. On the compute clusters' head node, every user account has their own home directory. These are the private spaces of researchers and are referred to as $HOME in Fig. 28.1. Analyses should be conducted on the cluster's compute nodes ($COMPUTE). $HOME and $COMPUTE are not managed or trusted by data management personnel, and are seen as *ephemeral* (short-lived). The RIA store ($DATA) can be accessed both from $HOME and $COMPUTE, in both directions: Researchers can pull datasets from the store, push new datasets to it, or update (certain) existing datasets. $DATA is the one location in which experienced data management personnel ensures back-up and archival, performs house-keeping, and handles PERMISSIONS, and is thus were pristine raw data is stored or analysis code or results from $COMPUTE and $HOME should end up in. This aids organization, and allows a central management of back-ups and archival, potentially by data stewards or similar data management personnel with no domain knowledge about data contents.

One aspect of the problem are disk-space unaware computing workflows. Researchers make and keep numerous copies of data in their home directory and perform computationally expensive analyses on the headnode of a compute cluster because they do not know better, and/or want to do it in the easiest way possible. A general change for the better can be achieved by imposing sensible limitations and restrictions on what can be done at which scale: Data from the RIA store ($DATA) is accessible to researchers for exploration and computation, but the scale of the operations they want to perform can require different approaches. In their $HOME, researchers are free to do whatever they want as long as it is within the limits of their machines or their user accounts (100GB). Thus, researchers can explore data, test and develop code, or

Fig. 28.1: Trinity of research data handling: The data store ($DATA) is managed and backed-up. The compute cluster ($COMPUTE) has an analysis-appropriate structure with adequate resources, but just as users workstations/laptops ($HOME), it is not concerned with data hosting.

visualize results, but they can not create complete dataset copies or afford to keep an excess of unused data around. Only $COMPUTE has the necessary hardware requirements for expensive computations. Thus, within $HOME, researchers are free to explore data as they wish, but scaling requires them to use $COMPUTE. By using a job scheduler, compute jobs of multiple researchers are distributed fairly across the available compute infrastructure. Version controlled (and potentially reproducible) research logs and the results of the analyses can be pushed from COMPUTE to $DATA for back-up and archival, and hence anything that is relevant for a research project is tracked, backed-up, and stored, all without straining available disk-space on the cluster afterwards. While the imposed limitations are independent of DataLad, DataLad can make sure that the necessary workflows are simple enough for researchers of any seniority, background, or skill level.

Remote indexed archive (RIA) stores

A RIA store is a storage solution for DataLad datasets that can be flexibly extended with new datasets, independent of static file names or directory hierarchies, and that can be (automatically) maintained or queried without requiring expert or domain knowledge about the data. At its core, it is a flat, file-system based repository representation of any number of datasets, limited only by disk-space constrains of the machine it lies on.

Put simply, a RIA store is a dataset storage location that allows for access to and collaboration on DataLad datasets. The high-level workflow overview is as follows: Create a dataset, use the **datalad create-sibling-ria** command to establish a connection to an either pre-existing or not-yet-existing RIA store, publish dataset contents with **datalad push**, (let others) clone the dataset from the RIA store, and (let others) publish and pull updates. In the case of large, institute-wide datasets, a RIA store (or multiple RIA stores) can serve as a central storage location that enables fine-grained data access to everyone who needs it, and as a storage and back-up location for all analyses datasets. Beyond constituting central storage locations, RIA stores also ease dataset maintenance and queries: If all datasets of an institute are kept in a single RIA store, questions such as "Which projects use this data as their input?", "In which projects was the student with this Git identity involved?", "Give me a complete research log of what was done for this publication", or "Which datasets weren't used in the last 5 years?" can be answered automatically with Git tools, without requiring expert knowledge about the contents of any of the datasets, or access to the original creators of the dataset. To find out more about RIA stores, check out section *Remote Indexed Archives for dataset storage and backup* (page 294).

RIA store workflows

Configurations can hide the technical layers

Setting up a RIA store and appropriate siblings is fairly easy – it requires only the **datalad create-sibling-ria** command. However, in the institute this usecase describes, in order to spare users knowing about RIA stores, custom configurations are distributed via DataLad's run-procedures to simplify workflows further and hide the technical layers of the RIA setup:

A custom procedure[631] performs the relevant sibling setup with a fully configured link to the RIA store, and, on top of it, also creates an associated repository with a publication dependency

---

[631] https://jugit.fz-juelich.de/inm7/infrastructure/inm7-datalad/blob/master/inm7_datalad/resources/procedures/cfg_inm7.py

on the RIA store to an institute's GitLab instance[632]. With a procedure like this in place system-wide, an individual researcher only needs to call the procedure right at the time of dataset creation, and has a fully configured and set up analysis dataset afterwards:

```
$ datalad create -c inm7 <PATH>
```

Working in this dataset will require only **datalad save** and **datalad push** commands, and configurations ensure that the projects history and results are published where they need to be: The RIA store, for storing and archiving the project including data, and GitLab, for exposing the projects progress to the outside and ease collaboration or supervision. Users do not need to know the location of the store, its layout, or how it works – they can go about doing their science, while DataLad handles publications routines.

In order to get input data from datasets hosted in the datastore without requiring users to know about dataset IDs or construct `ria+` URLs, superdatasets get a SIBLING on GITLAB or GITHUB with a human readable name. Users can clone the superdatasets from the web hosting service, and obtain data via **datalad get**. A concrete example for this is described in the usecase *Scaling up: Managing 80TB and 15 million files from the HCP release* (page 433). While **datalad get** will retrieve file or subdataset contents from the RIA store, users will not need to bother where the data actually comes from.

Summary

The infrastructural and workflow changes around DataLad datasets in RIA stores improve the efficiency of the institute:

With easy local version control workflows and DataLad-based data management routines, researchers are able to focus on science and face barely any technical overhead for data management. As file content for analyses is obtained *on demand* via **datalad get**, researchers selectively obtain only those data they need instead of having complete copies of datasets as before, and thus save disk space. Upon **datalad push**, computed results and project histories can be pushed to the data store and the institute's GitLab instance, and be thus backed-up and accessible for collaborators or supervisors. Easy-to-reobtain input data can safely be dropped to free disk space on the compute cluster. Sensible incentives for computing and limitations on disk space prevent unmanaged clutter. With a RIA store full of bare git repositories, it is easily maintainable by data stewards or system administrators. Common compression or cleaning operations of Git and git-annex are performed without requiring knowledge about the data inside of the store, as are queries on interesting aspects of datasets, potentially across all of the datasets of the institute. With a remote data store setup, the compute cluster is efficiently used for computations instead of data storage. Researchers can not only compute their analyses faster and on larger datasets than before, but with DataLad's version control capabilities their work also becomes more transparent, open, and reproducible.

---

[632] To re-read about DataLad's run-procedures, check out section *Configurations to go* (page 126). You can find the source code of the procedure on GitLab[633].

[633] https://jugit.fz-juelich.de/inm7/infrastructure/inm7-datalad/blob/master/inm7_datalad/resources/procedures/cfg_inm7.py

USING GLOBUS AS A DATA STORE FOR THE CANADIAN OPEN NEUROSCIENCE PORTAL

This use case shows how the Canadian Open Neuroscience Portal (CONP)[634] disseminates data as DataLad datasets using the Globus[635] network with GIT-ANNEX, a custom git-annex SPECIAL REMOTE, and Datalad. It demonstrates

1. How to enable the git-annex Globus special remote[636] to access files content from Globus.org[637],

2. The workflows used to access datasets via the Canadian Open Neuroscience Portal (CONP)[638],

3. An example of disk-space aware computing with large datasets distributed across systems that avoids unnecessary replication, eased by DataLad and GIT-ANNEX.

29.1 The Challenge

Every day, researchers from different fields strive to advance present state-of-the-art scientific knowledge by generating and publishing novel results. Crucially, they must share such results with the scientific community to enable other researchers to further build on existing data and avoid duplicating work.

The Canadian Open Neuroscience Portal (CONP)[639] is a publicly available platform that aims to remove the technical barriers to practicing open science and improve the accessibility and reusability of neuroscience research to accelerate the pace of discovery. To this end, the platform will provide a unified interface that – among other things – enables sharing and open dissemination of both neuroscience data and methods to the global community. Managing the scientific data ecosystem is extremely challenging given the amount of new data generated every day, however. CONP must take a strategic solution to allow researchers to

- dynamically work on present data,

- upload new versions of the data, and

- generate additional scientific work.

---

[634] https://conp.ca/
[635] https://www.globus.org/
[636] https://github.com/CONP-PCNO/git-annex-remote-globus
[637] https://www.globus.org/
[638] https://conp.ca/
[639] https://conp.ca/

An underlying data management system to achieve this must be flexible, dynamic and light-weight. It would need to have the ability to easily distribute datasets across multiple locations to reduce the need of re-collecting or replicating data that is similar to already existing datasets.

29.2 The Datalad Approach

CONP makes use of Datalad as a data management tool to enable efficient analysis and work on datasets: Datalad minimizes the computational cost of holding full storage of datasets versions, it allows files in a dataset to be distributed across multiple download sources, and to be retrieved on demand only to save disk space. Therefore, it is common practice for researchers to both download and publish research content in a dataset format via a CONP, which provides them with a vast dataset repository.

> **M29.1 Basic principles of DataLad for new readers**
>
> If you are new to DataLad, the introduction of the handbook and the chapter *DataLad datasets* (page 34) can give you a good idea of what DataLad and its underlying tools can to, as well as a hands-on demonstration. This findoutmore, in the meantime, sketches a high-level overview of the principles behind DataLad's data sharing capacities.
> Datalad is built on top of Git[640] and git-annex[641], and enables data version control. A one-page overview can be found in section *What you really need to know* (page 27). GIT-ANNEX is a useful tool that extends Git with the ability to manage repositories in a lightweight fashion even if they contain large amounts of data. One main principle of git-annex lies storing data that should not be stored in Git (e.g., due to size limits) in an ANNEX. In its place, it generates symbolic links (SYMLINKs) to these *annexed* files that encode their file content. Only the symlinks are committed into GIT while GIT-ANNEX handles data management in the annex. A detailed explanation of this process can be found in the section *Data integrity* (page 85), but the outcome of it is a light-weight Git repository that can be cloned fast and yet contains access to arbitrarily large data managed by GIT-ANNEX.
> In the case of data sharing procedures, annexed data can be stored in various third party hosting services configured as special remotes[642]. When retrieving data, GIT-ANNEX requests access to the primary data source storing those files to retrieve actual files content when the user needs it.
>
> ---
> [640] https://git-scm.com/
> [641] https://git-annex.branchable.com/
> [642] https://git-annex.branchable.com/special_remotes/

The workflows for users to get data are straightforward: Users log into the CONP portal and install Datalad datasets with `datalad install -r <dataset>`. This gives them access to the annexed files (as mentioned in the findoutmore above, large files replaced by their symlinks). To request the content of the annexed files, they simply download those files locally in their filesystem using `datalad get path/to/file`. So simple!

On a technical level, under the hood, GIT-ANNEX needs to have a connection established with the primary data source, the SPECIAL REMOTE, that hosts and provides the requested files' contents. In some cases, annexed files are stored in Globus.org[643]. Globus is an efficient transfer files system suitable for researchers to share and transfer files between so called *endpoints*, locations in Globus.org where files get uploaded by their owners or get transferred to, that can be either

---
[643] https://www.globus.org/

private or public. Annexed file contents are stored in such Globus endpoints[644]. Therefore, when users download annexed files, Globus communicates with git-annex to provide access to files content. Given this functionality, we can say that Globus works as a data store for git-annex, or in technical terms, that Globus is configured to work as a SPECIAL REMOTE for git-annex. This is possible via the git-annex backend interface implementation for Globus called git-annex-globus-remote[645] developed by CONP. In conjunction, CONP and the git-annex-globus-remote constitute the building blocks that enable access to datasets and its data: CONP hosts small-sized datasets, and Globus.org is the data store that (large) file content can be retrieved from.

To sum up, CONP makes a variety of datasets available and provides them to researchers as Datalad datasets that have the regular, advantageous Datalad functionality. All of this exists thanks to the ability of git-annex and Datalad to interface with special remote locations across the web such as Globus.org[646] to request access to data. In this way, researchers have access to a wide research data ecosystem and can use and reuse existing data, thus reducing the need of data replication.

29.3 Step-by-Step

Globus as git-annex data store

A remote data store exists thanks to git-annex (which DataLad builds upon): git-annex uses a key-value pair to reference files. In the git-annex object tree, large files in datasets are stored as values while the key is generated from their contents and is checked into Git. The key is used to reference the location of the value in the object tree[653]. The OBJECT-TREE (or keystore) with the data contents can be located anywhere – its location only needs to be encoded using a special remote. Therefore, thanks to the git-annex-globus-remote[647] interface, Globus.org provides git-annex with location information to retrieve values and access files content with the corresponding keys. To ultimately enable end users' access to data, git-annex registers Globus locations by assigning them to Globus-specific URLs, such as `globus://dataset_id/path/to/file`. Each Globus URL is associated with a the key corresponding to the given file. The use of a Globus URL protocol is a fictitious mean to assign each file of the dataset a unique location and source and therefore, it is a wrapper for additional validation that is performed by the git-annex-globus-remote to check on the actual presence of the file within the Globus transfer file ecosystem. In other words, the 'Globus URL' is simply an alias of an existing file located on the web and specifically available in Globus.org. Registration of Globus URLs in git-annex is among the configuration procedures carried out on an administrative, system-wide level, and users will only deal with direct easy access of desired files.

With this, Globus is configured to receive data access requests from git-annex and to respond back if data is available. Currently, the git-annex-globus-remote only supports data *download* operations. In the future, it could be useful for additional functionality as well. When the globus special remote gets initialized for the first time, the user has to authenticate to Globus.org using ORCID[648] , Gmail[649] or a specific Globus account. This step will enable git-annex to then initial-

---

[644] https://docs.globus.org/faq/globus-connect-endpoints/#what_is_an_endpoint

[645] https://github.com/CONP-PCNO/git-annex-remote-globus

[646] https://www.globus.org

[653] More details on how GIT-ANNEX handles data underneath the hood and how the OBJECT-TREE works can be found in section *Data integrity* (page 85).

[647] https://github.com/CONP-PCNO/git-annex-remote-globus

[648] https://orcid.org/

[649] https://mail.google.com

---

ize the globus special remote and establish the communication process. Instructions to use the globus special remote are available at github.com/CONP-PCNO/git-annex-remote-globus[650]. Guidelines specifying the standard communication protocol to implement a custom special remote can be found at git-annex.branchable.com/design/external_special_remote_protocol[651].

An example using Globus from a user perspective

It always starts with a dataset, installed with either **datalad install** or **datalad clone**.

```
$ datalad install -r <dataset>
$ cd <dataset>
```

In order to get access to annexed data stored on Globus.org, users need to install the globus-special-remote. If it is the first time using Globus, users will need to authenticate to Globus.org by running the git-annex-remote-globus setup command:

```
$ pip install git-annex-remote-globus
# if first time
$ git-annex-remote-globus setup
```

After the installation of a dataset, we can see that most of the files in the dataset are annexed: Listing a file with ls -l will reveal a SYMLINK to the dataset's annex.

```
$ ls -l NeuroMap_data/cortex/mask/mask.mat
 cortex/mask/mask.mat -> ../../../.git/annex/objects/object.mat
```

However, without having any content downloaded yet, the symlink currently points into a void, and tools will not be able to open the file as its contents are not yet locally available.

```
$ cat NeuroMap_data/cortex/mask/mask.mat
  NeuroMap_data/cortex/mask/mask.mat: No such file or directory
```

However, data retrieval is easy. At first, users have to enable the globus remote.

```
$ git annex enableremote globus
 enableremote globus ok
 (recording state in git...)
```

After that, they can download any file, directory, or complete dataset using **datalad get**:

```
$ datalad get NeuroMap_data/cortex/mask/mask.mat
 get(ok): NeuroMap_data/cortex/mask/mask.mat (file) [from globus...]

$ ls -l NeuroMap_data/cortex/mask/mask.mat
 cortex/mask/mask.mat -> ../../../.git/annex/objects/object.mat

$ cat NeuroMap_data/cortex/mask/mask.mat
 # you can now access the file !
```

Downloaded! Researchers could now use this dataset to replicate previous analyses and further build on present data to bring scientific knowledge forward. CONP thus makes a variety of datasets flexibly available and helps to disseminate data. The on-demand availability of files in

---

[650] https://github.com/CONP-PCNO/git-annex-remote-globus
[651] https://git-annex.branchable.com/design/external_special_remote_protocol/

datasets can help scientists to save disk space. For this, they could get only those data files that they need instead of obtaining complete copies of the dataset, or they could locally **drop** data that is hosted and thus easily re-available on Globus.org after their analyses are done.

29.4 Resources

The README at github.com/CONP-PCNO/git-annex-remote-globus[652] provides an excellent and in-depth overview of how to install and use the git-annex special remote for Globus.org.

---

[652] https://github.com/CONP-PCNO/git-annex-remote-globus

DATALAD FOR REPRODUCIBLE MACHINE-LEARNING ANALYSES

This use case demonstrates an automatically and computationally reproducible analyses in the context of a machine learning (ML) project. It demonstrates on an example image classification analysis project how one can

- link data, models, parametrization, software and results using `datalad containers-run`

- keep track of results and compare them across models or parametrizations

- stay computationally reproducible, transparent, and importantly, intuitive and clear

30.1 The Challenge

Chad is a recent college graduate and has just started in a wicked start-up that prides itself with using "AI and ML for individualized medicine" in the Bay area. Even though he's extraordinarily motivated, the fast pace and pressure to deliver at his job are still stressful. For his first project, he's tasked with training a machine learning model to detect cancerous tissue in computer tomography (CT)[654] images. Excited and eager to impress, he builds his first image classification ML model with state of the art Python libraries and a stochastic gradient descent (SGD)[655] classifier. "Not too bad", he thinks, when he shares the classification accuracy with his team lead, "way higher than chance level!" "Fantastic, Chad, but listen, we really need a higher accuracy than this. Our customers deserve that. Turn up the number of iterations. Also, try a random forest classification instead. And also, I need that done by tomorrow morning latest, Chad. Take a bag of organic sea-weed-kale crisps from the kitchen, oh, and also, you're coming to our next project pitch at the roof-top bar on Sunday?"

Hastily, Chad pulls an all-nighter to adjust his models by dawn. Increase iterations here, switch classifier there, oh no, did this increase or decrease the overall accuracy? Tune some parameters here and there, re-do that previous one just one more time just to be sure. A quick two-hour nap on the office couch, and he is ready for the daily scrum[656] in the morning. "Shit, what accuracy belonged to which parametrization again?", he thinks to himself as he pitches his analysis and presents his results. But everyone rushes to the next project already.

A week later, when a senior colleague is tasked with checking his analyses, Chad needs to spend a few hours with them to them guide through his chaotic analysis directory full of jupyter notebooks. They struggle to figure out which Python libraries to install on the colleagues computer, have to adjust hard-code ABSOLUTE PATHs, and fail to reproduce the results that he presented.

---

[654] https://en.wikipedia.org/wiki/CT_scan
[655] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[656] https://en.wikipedia.org/wiki/Scrum_(software_development)#Daily_scrum

30.2 The DataLad Approach

Machine learning analyses are complex: Beyond data preparation and general scripting, they typically consist of training and optimizing several different machine learning models and comparing them based on performance metrics. This complexity can jeopardize reproducibility – it is hard to remember or figure out which model was trained on which version of what data and which has been the ideal optimization. But just like any data analysis project, machine learning projects can become easier to understand and reproduce if they are intuitively structured, appropriately version controlled, and if analysis executions are captured with enough (ideally machine-readable and re-executable) provenance.

DataLad has many concepts and tools that assist in creating transparent and computationally and automatically reproducible analyses. From general principles on how to structure analyses projects to linking and versioning software and data alongside to code or capturing analysis execution as re-executable run-records. To make a machine-learning project intuitively structured and transparent, Chad applies DataLad's YODA principles to his work. He keeps the training and testing data a reusable, standalone component, installed as a subdataset, and keeps his analysis dataset completely self-contained with RELATIVE PATHs in all his scripts. Later, he can share his dataset without the need to adjust paths. Chad also attaches a software container to his dataset, so that others don't need to recreate his Python environment. And lastly, he wraps every command that he executes in a `datalad containers-run` call, such that others don't need to rely on his brain to understand the analysis, but can have a computer recompute every analysis step in the correct software environment. Using concise commit messages and TAGs, Chad creates a transparent and intuitive dataset history. With these measures in place, he can experiment flexibly with various models and data, and does not only have means to compare his models, but can also set his dataset to the state in which his most preferred model is ready to be used.

30.3 Step-by-Step

Required software

The analysis requires the Python packages scikit-learn[657], scikit-image[658], pandas[659], and numpy[660]. We have build a SINGULARITY SOFTWARE CONTAINER with all relevant software, and the code below will use the `datalad-containers` extension[672] to download the container from SINGULARITY-HUB and execute all analysis in this software environment. If you do not want to install the `datalad-containers` extension or Singularity, you can also create a VIRTUAL ENVIRONMENT with all necessary software if you prefer[673], and exchange the datalad

---

[657] https://scikit-learn.org/stable/

[658] https://scikit-image.org/

[659] https://pandas.pydata.org/

[660] https://numpy.org/

[672] You can install the `datalad-containers` extension from PIP via `pip install datalad-container`. You can find out more about extensions in general in the section *DataLad extensions* (page 285), and you can more computationally reproducible analysis using `datalad container` in the chapter *Computational reproducibility with software containers* (page 166) and the usecase *An automatically and computationally reproducible neuroimaging analysis from scratch* (page 420).

[673] Unsure how to create a VIRTUAL ENVIRONMENT? You can find a tutorial using PIP and the `virtualenv` module in the Python docs[674].

[674] https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/

containers-run commands below with `datalad run` commands.

Let's start with an overview of the analysis plans: We're aiming for an image classification analysis. In this type of ML analysis, a *classifier* is trained on a subset of data, the *training set*, and is then used for predictions on a previously unseen subset of data, the *test set*. Its task is to label the test data with one of several class attributes it is trained to classify, such as "cancerous" or "non-cancerous" with medical data[661], "cat" or "dog"[662] with your pictures of pets, or "spam" versus "not spam" in your emails. In most cases, classification analyses are *supervised* learning methods: The correct class attributes are known, and the classifier is tested on a *labeled* set of training data. Its classification accuracy is calculated from comparing its performance on the unlabeled testing set with its correct labels. As a first analysis step, train and testing data therefore need to be labeled – both to allow model training and model evaluation. In a second step, a classifier needs to be trained on the labeled test data. It learns which features are to be associated with which class attribute. In a final step, the trained classifier classifies the test data, and its results are evaluated against the true labels.

Below, we will go through a image classification analysis on a few categories in the Imagenette dataset[663], a smaller subset of the Imagenet dataset[664], one of the most widely used large scale dataset for bench-marking Image Classification algorithms. It contains images from ten categories (tench (a type of fish), English springer (a type of dog), cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute). We will prepare a subset of the data, and train and evaluate different types of classifier. The analysis is based on this tutorial[665].

First, let's create an input data dataset. Later, this dataset will be installed as a subdataset of the analysis. This complies to the *YODA principles* (page 136) and helps to keep the input data modular, reusable, and transparent.

```
$ datalad create imagenette
[INFO] Creating a new annex repo at /home/me/usecases/imagenette
create(ok): /home/me/usecases/imagenette (dataset)
```

The original Imagenette dataset contains 10 image categories can be downloaded as an archive from Amazon (s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.tgz[666]), but for this tutorial we're using a subset of this dataset with only two categories. It is available as an archive from the OPEN SCIENCE FRAMEWORK (OSF). The **datalad download-url --archive** not only extracts and saves the data, but also registers the datasets origin such that it can re-retrieved on demand from its original location.

```
$ cd imagenette
# 0.12.2 <= datalad < 0.13.4  needs the configuration option -c datalad.runtime.use-
↪patool=1 to handle .tgz
$ datalad download-url \
  --archive \
  --message "Download Imagenette dataset" \
  'https://osf.io/d6qbz/download'
[INFO] Downloading 'https://osf.io/d6qbz/download' into '/home/me/usecases/imagenette/'
```

<div align="right">(continues on next page)</div>

---

[661] https://www.nature.com/articles/d41586-020-00847-2
[662] https://www.kaggle.com/c/dogs-vs-cats
[663] https://github.com/fastai/imagenette
[664] http://www.image-net.org/
[665] https://realpython.com/python-data-version-control/
[666] https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.tgz

```
[INFO] Adding content of the archive /home/me/usecases/imagenette/imagenette2-160.tgz␣
↪into annex AnnexRepo(/home/me/usecases/imagenette)
[INFO] Initiating special remote datalad-archives
[INFO] Finished adding /home/me/usecases/imagenette/imagenette2-160.tgz: Files processed:␣
↪2701, +annex: 2701
[INFO] Finished extraction
download_url(ok): /home/me/usecases/imagenette/imagenette2-160.tgz (file)
save(ok): . (dataset)
add-archive-content(ok): /home/me/usecases/imagenette (dataset)
action summary:
  add (ok: 1)
  add-archive-content (ok: 1)
  download_url (ok: 1)
  save (ok: 1)
```

Next, let's create an analysis dataset. For a pre-structured and pre-configured starting point, the dataset can be created with the yoda and `text2git` RUN PROCEDURES[675]. These configurations create a code/ directory, place some place-holding README files in appropriate places, and make sure that all text files, e.g. scripts or evaluation results, are kept in GIT to allow for easier modifications.

> **W30.1 Note for Windows-Users**
>
> Hey there! If you are using **Windows 10** (not Windows Subsystem for Linux (WSL)[667]) **without the custom-built git-annex** installer mentioned in the installation section, you need a work-around.
> Instead of running `datalad create -c text2git -c yoda ml-project`, please remove the configuration `-c text2git` from the command and run only `datalad create -c yoda ml-project`:
>
> ```
> $ datalad create -c yoda ml-project
> [INFO] Creating a new annex repo at C:\Users\mih\ml-project
> [INFO] Detected a filesystem without fifo support.
> [INFO] Disabling ssh connection caching.
> [INFO] Detected a crippled filesystem.
> [INFO] Scanning for unlocked files (this may take some time)
> [INFO] Entering an adjusted branch where files are unlocked as this filesystem does␣
> ↪not support locked files.
> [INFO] Switched to branch 'adjusted/master(unlocked)'
> [INFO] Running procedure cfg_yoda
> [INFO] == Command start (output follows) =====
> [INFO] == Command exit (modification check follows) =====
> create(ok): C:\Users\mih\ml-project (dataset)
> ```
>
> Instead of the `text2git` configuration, you need to create a configuration by hand by pasting the following lines of text into the (hidden) `.gitattributes` file in your newly created dataset. *Tuning datasets to your needs* (page 112) can explain the details of this procedure.
> Here are lines that need to be appended to the existing lines in `.gitattributes` and will mimic the configuration `-c text2git` would apply:
>
> ```
> *.json annex.largefiles=nothing
> ```
>
> You can achieve this by copy-pasting the following code snippets into your terminal (but

---

[675] To re-read about RUN PROCEDUREs, check out section *Configurations to go* (page 126).

> you can also add them using a text editor of your choice):
>
> ```
> $ echo\ >> .gitattributes && echo *.json annex.largefiles=nothing >> .gitattributes
> ```
>
> Afterwards, these should be the contents of `.gitattributes`:
>
> ```
> $ cat .gitattributes
>   * annex.backend=MD5E
>   **/.git* annex.largefiles=nothing
>   CHANGELOG.md annex.largefiles=nothing
>   README.md annex.largefiles=nothing
>   *.json annex.largefiles=nothing
> ```
>
> Lastly, run this piece of code to save your changes:
>
> ```
> $ datalad save -m "Windows-workaround: custom config to place text into Git" .
> ↪gitattributes
> ```
>
> ───────────────────
> [667] https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux

```
$ cd ../
$ datalad create -c text2git -c yoda ml-project
[INFO] Creating a new annex repo at /home/me/usecases/ml-project
[INFO] Running procedure cfg_text2git
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
[INFO] Running procedure cfg_yoda
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
create(ok): /home/me/usecases/ml-project (dataset)
```

Afterwards, the input dataset can be installed from a local path as a subdataset, using **datalad clone** with the -d/--dataset flag and a . to denote the current dataset:

```
$ cd ml-project
$ mkdir -p data
# install the dataset into data/
$ datalad clone -d . ../imagenette data/raw
[INFO] Cloning dataset to Dataset(/home/me/usecases/ml-project/data/raw)
[INFO] Attempting to clone from ../imagenette to /home/me/usecases/ml-project/data/raw
[INFO] Completed clone attempts for Dataset(/home/me/usecases/ml-project/data/raw)
[INFO] scanning for annexed files (this may take some time)
install(ok): data/raw (dataset)
add(ok): data/raw (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
action summary:
  add (ok: 3)
  install (ok: 1)
  save (ok: 2)
```

Here are the dataset contents up to now:

```
# show the directory hierarchy
```

```
$ tree -d
.
├── code
└── data
    └── raw
        ├── train
        │   ├── n03445777
        │   └── n03888257
        └── val
            ├── n03445777
            └── n03888257

9 directories
```

Next, let's add the necessary software to the dataset. This is done using the `datalad containers` extension and the **datalad container-add** command. This command takes an arbitrary name and a path or url to a SOFTWARE CONTAINER, registers the containers origin, and adds it under the specified name to the dataset. If used with a public url, for example to SINGULARITY-HUB, others that you share your dataset with can retrieve the container as well[672].

```
$ datalad containers-add software --url shub://adswa/python-ml:1
[INFO] Initiating special remote datalad
add(ok): .datalad/config (file)
save(ok): . (dataset)
containers_add(ok): /home/me/usecases/ml-project/.datalad/environments/software/image␣
↪(file)
action summary:
  add (ok: 1)
  containers_add (ok: 1)
  save (ok: 1)
```

At this point, with input data and software set-up, we can start with the first step: Dataset preparation. The imagenette dataset is structured in `train/` and `val/` folder, and each folder contains one sub-folder per image category. To prepare the dataset for training and testing a classifier, we create a mapping between file names and image categories.

In this example we only use two categories, "golf balls" (subdirectory `n03445777`) and "parachutes" (subdirectory `n03888257`). The following script creates two files, `data/train.csv` and `data/test.csv` from the input data. Each contains file names and category associations for the files in those subdirectories. Note how, in accordance to the *YODA principles* (page 136), the script only contains RELATIVE PATHs to make the dataset portable.

```
$ cat << EOT > code/prepare.py
#!/usr/bin/env python3

import pandas as pd
from pathlib import Path

FOLDERS_TO_LABELS = {"n03445777": "golf ball",
                     "n03888257": "parachute"}


def get_files_and_labels(source_path):
    images = []
    labels = []
```

```
    for image_path in source_path.rglob("*/*.JPEG"):
        filename = image_path
        folder = image_path.parent.name
        if folder in FOLDERS_TO_LABELS:
            images.append(filename)
            label = FOLDERS_TO_LABELS[folder]
            labels.append(label)
    return images, labels


def save_as_csv(filenames, labels, destination):
    data_dictionary = {"filename": filenames, "label": labels}
    data_frame = pd.DataFrame(data_dictionary)
    data_frame.to_csv(destination)


def main(repo_path):
    data_path = repo_path / "data"
    train_path = data_path / "raw/train"
    test_path = data_path / "raw/val"
    train_files, train_labels = get_files_and_labels(train_path)
    test_files, test_labels = get_files_and_labels(test_path)
    save_as_csv(train_files, train_labels, data_path / "train.csv")
    save_as_csv(test_files, test_labels, data_path / "test.csv")


if __name__ == "__main__":
    repo_path = Path(__file__).parent.parent
    main(repo_path)
EOT
```

Executing the here document[668] in the code block above has created a script code/prepare.py:

```
$ datalad status
untracked: code/prepare.py (file)
```

We add it to the dataset using **datalad save**:

```
$ datalad save -m "Add script for data preparation for 2 categories" code/prepare.py
add(ok): code/prepare.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

This script can now be used to prepare the data. Note how it, in accordance to the *YODA principles* (page 136), saves the files into the superdataset, and leaves the input dataset untouched. When ran, it will create files with the following structure:

```
,filename,label
0,data/raw/imagenette2-160/val/n03445777/n03445777_20061.JPEG,golf ball
1,data/raw/imagenette2-160/val/n03445777/n03445777_9740.JPEG,golf ball
2,data/raw/imagenette2-160/val/n03445777/n03445777_3900.JPEG,golf ball
3,data/raw/imagenette2-160/val/n03445777/n03445777_5862.JPEG,golf ball
```

---

[668] https://en.wikipedia.org/wiki/Here_document

```
4,data/raw/imagenette2-160/val/n03445777/n03445777_4172.JPEG,golf ball
5,data/raw/imagenette2-160/val/n03445777/n03445777_14301.JPEG,golf ball
6,data/raw/imagenette2-160/val/n03445777/n03445777_2951.JPEG,golf ball
7,data/raw/imagenette2-160/val/n03445777/n03445777_8732.JPEG,golf ball
8,data/raw/imagenette2-160/val/n03445777/n03445777_5810.JPEG,golf ball
9,data/raw/imagenette2-160/val/n03445777/n03445777_3132.JPEG,golf ball
[...]
```

To capture all provenance and perform the computation in the correct software environment, this is best done in a **datalad containers-run** command:

```
$ datalad containers-run -n software \
  -m "Prepare the data for categories golf balls and parachutes" \
  --input 'data/raw/train/n03445777' \
  --input 'data/raw/val/n03445777' \
  --input 'data/raw/train/n03888257' \
  --input 'data/raw/val/n03888257' \
  --output 'data/train.csv' \
  --output 'data/test.csv' \
  "python3 code/prepare.py"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
get(ok): data/raw/train/n03445777 (directory)
get(ok): data/raw/val/n03445777 (directory)
get(ok): data/raw/val/n03888257 (directory)
get(ok): data/raw/train/n03888257 (directory)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 2, ok: 2704)
  save (notneeded: 1, ok: 1)
```

Beyond the script execution and container name (-n/--container-name), this command can take a human readable commit message to summarize the operation (-m/--message) and input and output specifications (-i/--input, -o/--output). DataLad will make sure to retrieve everything labeled as --input prior to running the command, and specifying --output ensures that the files can be updated should the command be reran at a later point[676]. It saves the results of this command together with a machine-readable run-record into the dataset history.

Next, the first model can be trained.

```
$ cat << EOT > code/train.py
#!/usr/bin/env python3

from joblib import dump
from pathlib import Path

import numpy as np
import pandas as pd
from skimage.io import imread_collection
from skimage.transform import resize
from sklearn.linear_model import SGDClassifier
```

---

[676] The chapter *DataLad, Run!* (page 59) introduces the options of datalad run and demonstrates their use. Note that --outputs don't need to be individual files, but could also be directories or GLOBBING terms.

```python
def load_images(data_frame, column_name):
    filelist = data_frame[column_name].to_list()
    image_list = imread_collection(filelist)
    return image_list


def load_labels(data_frame, column_name):
    label_list = data_frame[column_name].to_list()
    return label_list


def preprocess(image):
    resized = resize(image, (100, 100, 3))
    reshaped = resized.reshape((1, 30000))
    return reshaped


def load_data(data_path):
    df = pd.read_csv(data_path)
    labels = load_labels(data_frame=df, column_name="label")
    raw_images = load_images(data_frame=df, column_name="filename")
    processed_images = [preprocess(image) for image in raw_images]
    data = np.concatenate(processed_images, axis=0)
    return data, labels


def main(repo_path):
    train_csv_path = repo_path / "data/train.csv"
    train_data, labels = load_data(train_csv_path)
    sgd = SGDClassifier(max_iter=10)
    trained_model = sgd.fit(train_data, labels)
    dump(trained_model, repo_path / "model.joblib")


if __name__ == "__main__":
    repo_path = Path(__file__).parent.parent
    main(repo_path)
EOT
```

This script trains a stochastic gradient descent classifier on the training data. The files in the
train.csv file a read, preprocessed into the same shape, and an SGD model is fitted to the
predict the image labels from the data. The trained model is then saved into a model.joblib
file – this allows to transparently cache the classifier as a Python object to disk. Later, the cached
model can be applied to various data with the need to retrain the classifier[669]. Let's save the
script.

```
$ datalad save -m "Add SGD classification script" code/train.py
add(ok): code/train.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

---

[669] https://scikit-learn.org/stable/modules/model_persistence.html

The last analysis step needs to test the trained classifier. We will use the following script for this:

```
$ cat << EOT > code/evaluate.py

#!/usr/bin/env python3

from joblib import load
import json
from pathlib import Path

from sklearn.metrics import accuracy_score

from train import load_data


def main(repo_path):
    test_csv_path = repo_path / "data/test.csv"
    test_data, labels = load_data(test_csv_path)
    model = load(repo_path / "model.joblib")
    predictions = model.predict(test_data)
    accuracy = accuracy_score(labels, predictions)
    metrics = {"accuracy": accuracy}
    print(metrics)
    accuracy_path = repo_path / "accuracy.json"
    accuracy_path.write_text(json.dumps(metrics))


if __name__ == "__main__":
    repo_path = Path(__file__).parent.parent
    main(repo_path)
EOT
```

It will load the trained and dumped model and use it to test its prediction performance on the yet unseen test data. To evaluate the model performance, it calculates the accuracy of the prediction, i.e., the proportion of correctly labeled images, prints it to the terminal, and saves it into a json file in the superdataset. As this script constitutes the last analysis step, let's save it with a TAG. Its entirely optional to do this, but just as commit messages are an easier way for humans to get an overview of a commits contents, a tag is an easier way for humans to identify a change than a commit hash. With this script set up, we're ready for analysis, and thus can tag this state `ready4analysis` to identify it more easily later.

```
$ datalad save -m "Add script to evaluate model performance" --version-tag "ready4analysis
↪" code/evaluate.py
add(ok): code/evaluate.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Afterwards, we can train the first model:

```
$ datalad containers-run -n software \
  -m "Train an SGD classifier on the data" \
  --input 'data/raw/train/n03445777' \
  --input 'data/raw/train/n03888257' \
```

```
  --output 'model.joblib' \
  "python3 code/train.py"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:573:␣
↪ConvergenceWarning: Maximum number of iteration reached before convergence. Consider␣
↪increasing max_iter to improve the fit.
  ConvergenceWarning)
[INFO] == Command exit (modification check follows) =====
save(ok): . (dataset)
action summary:
  add (ok: 1)
  get (notneeded: 4)
  save (notneeded: 1, ok: 1)
```

And finally, we're ready to find out how well the model did and run the last script:

```
$ datalad containers-run -n software \
  -m "Evaluate SGD classifier on test data" \
  --input 'data/raw/val/n03445777' \
  --input 'data/raw/val/n03888257' \
  --output 'accuracy.json' \
  "python3 code/evaluate.py"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
{'accuracy': 0.7515842839036755}
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 4)
  save (notneeded: 1, ok: 1)
```

Now this initial accuracy isn't yet fully satisfying. What could have gone wrong? The model
would probably benefit from a few more training iterations for a start. Instead of 10, the
patch below increases the number of iterations to 100. Note that the code block below does this
change with the stream editor SED for the sake of automatically executed code in the handbook,
but you could also apply this change with a text editor "by hand".

```
$ sed -i 's/SGDClassifier(max_iter=10)/SGDClassifier(max_iter=100)/g' code/train.py
```

Here's what has changed:

```
$ git diff
diff --git a/code/train.py b/code/train.py
index 3b309e1..017a6bf 100644
--- a/code/train.py
+++ b/code/train.py
@@ -39,7 +39,7 @@ def load_data(data_path):
 def main(repo_path):
     train_csv_path = repo_path / "data/train.csv"
     train_data, labels = load_data(train_csv_path)
-    sgd = SGDClassifier(max_iter=10)
+    sgd = SGDClassifier(max_iter=100)
     trained_model = sgd.fit(train_data, labels)
```

```
    dump(trained_model, repo_path / "model.joblib")
```

Let's save the change...

```
$ datalad save -m "Increase the amount of iterations to 100" --version-tag "SGD-100" code/
↪train.py
add(ok): code/train.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

... and try again.

As we need to retrain the classifier and re-evaluate its performance, we rerun every run-record between the point in time we created the SGD tag and now. This will update both the model. joblib and the accuracy.json files, but their past versions are still in the dataset history. One was to do this is to specify a range between the two tags, but likewise, commit hashes would work, or a specification using --since[677].

```
$ datalad rerun -m "Recompute classification with more iterations" ready4analysis..SGD-100
[INFO] run commit 08f7604; (Train an SGD clas...)
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): model.joblib (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): model.joblib (file)
save(ok): . (dataset)
[INFO] run commit 961a5be; (Evaluate SGD clas...)
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
{'accuracy': 0.7858048162230672}
[INFO] == Command exit (modification check follows) =====
add(ok): accuracy.json (file)
add(ok): code/__pycache__/train.cpython-37.pyc (file)
save(ok): . (dataset)
[INFO] skip-or-pick commit a729a18; a729a18 does not have a command; skipping or cherry␣
↪picking
run(ok): /home/me/usecases/ml-project (dataset) [a729a18 does not have a command;␣
↪skipping]
action summary:
  add (ok: 3)
  get (notneeded: 8)
  run (ok: 1)
  save (notneeded: 2, ok: 2)
  unlock (notneeded: 2, ok: 1)
```

Any better? Mhh, not so much. Maybe a different classifier does the job better. Let's switch from SGD to a random forest classification[670]. The code block below writes the relevant changes (highlighted) into the script.

---

[677] In order to re-execute any run-record in the last five commits, you could use --since=HEAD~5, for example. You could also, however, rerun the previous run commands sequentially, with datalad rerun <commit-hash>.

[670] https://en.wikipedia.org/wiki/Random_forest

```
$ cat << EOT >| code/train.py
#!/usr/bin/env python3

from joblib import dump
from pathlib import Path

import numpy as np
import pandas as pd
from skimage.io import imread_collection
from skimage.transform import resize
from sklearn.ensemble import RandomForestClassifier

def load_images(data_frame, column_name):
    filelist = data_frame[column_name].to_list()
    image_list = imread_collection(filelist)
    return image_list

def load_labels(data_frame, column_name):
    label_list = data_frame[column_name].to_list()
    return label_list

def preprocess(image):
    resized = resize(image, (100, 100, 3))
    reshaped = resized.reshape((1, 30000))
    return reshaped

def load_data(data_path):
    df = pd.read_csv(data_path)
    labels = load_labels(data_frame=df, column_name="label")
    raw_images = load_images(data_frame=df, column_name="filename")
    processed_images = [preprocess(image) for image in raw_images]
    data = np.concatenate(processed_images, axis=0)
    return data, labels

def main(repo_path):
    train_csv_path = repo_path / "data/train.csv"
    train_data, labels = load_data(train_csv_path)
    rf = RandomForestClassifier()
    trained_model = rf.fit(train_data, labels)
    dump(trained_model, repo_path / "model.joblib")

if __name__ == "__main__":
    repo_path = Path(__file__).parent.parent
    main(repo_path)
EOT
```

We need to save this change:

```
$ datalad save -m "Switch to random forest classification" --version-tag "random-forest"␣
↪code/train.py
add(ok): code/train.py (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

And now we can retrain and reevaluate again. This time, in order to have very easy access to

the trained models and results of the evaluation, we're rerunning the sequence of run-records in a new BRANCH[678]. This way, we have access to a trained random-forest model or a trained SGD model or their respective results by simply switching branches.

```
$ datalad rerun --branch="randomforest" -m "Recompute classification with random forest␣
↪classifier" ready4analysis..SGD-100
[INFO] checkout commit a254f4f;
[INFO] run commit 08f7604; (Train an SGD clas...)
[INFO] Making sure inputs are available (this may take some time)
unlock(ok): model.joblib (file)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
add(ok): model.joblib (file)
save(ok): . (dataset)
[INFO] run commit 961a5be; (Evaluate SGD clas...)
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
{'accuracy': 0.8124207858048162}
[INFO] == Command exit (modification check follows) =====
add(ok): accuracy.json (file)
add(ok): code/__pycache__/train.cpython-37.pyc (file)
save(ok): . (dataset)
[INFO] skip-or-pick commit a729a18; a729a18 does not have a command; skipping or cherry␣
↪picking
run(ok): /home/me/usecases/ml-project (dataset) [a729a18 does not have a command;␣
↪skipping]
action summary:
  add (ok: 3)
  get (notneeded: 8)
  run (ok: 1)
  save (notneeded: 2, ok: 2)
  unlock (notneeded: 2, ok: 1)
```

This updated the model.joblib file to a trained random forest classifier, and also updated accuracy.json with the current models' evaluation. The difference in accuracy between models could now for example be compared with a git diff of the contents of accuracy.json to the MASTER BRANCH:

```
$ git diff master -- accuracy.json
diff --git a/accuracy.json b/accuracy.json
index 9ff7e7c..f6e7ded 100644
--- a/accuracy.json
+++ b/accuracy.json
@@ -1 +1 @@
-{"accuracy": 0.7858048162230672}
\ No newline at end of file
+{"accuracy": 0.8124207858048162}
\ No newline at end of file
```

And if you decide to rather do more work on the SGD classier, you can go back to the previous MASTER BRANCH:

---

[678] Rerunning on a different BRANCH is optional but handy. Alternatively, you could checkout a previous state in the datasets history to get access to a previous version of a file, reset the dataset to a previous state, or use commands like `git cat-file` to read out a non-checked-out file. The section *Back and forth in time* (page 245) summarizes a number of common Git operations to interact with the dataset history.

```
$ git checkout master
$ cat accuracy.json
Switched to branch 'master'
{"accuracy": 0.7858048162230672}
```

Your Git history becomes a log of everything you did as well as the chance to go back to and forth between analysis states. And this is not only useful for yourself, but it makes your analyses and results also transparent to others that you share your dataset with. If you cache your trained models, there is no need to retrain them when traveling to past states of your dataset. And if any aspect of your dataset changes – from changes to the input data to changes to your trained model or code – you can rerun these analysis stages automatically. The attached software container makes sure that your analysis will always be rerun in the correct software environment, even if the dataset is shared with collaborators with systems that lack a Python installation.

30.4 References

The analysis is adapted from the chapter *Reproducible machine learning analyses: DataLad as DVC* (page 358), which in turn is based on this tutorial at RealPython.org[671].

---

[671] https://realpython.com/python-data-version-control/

CONTRIBUTING

If you are using DataLad for a use case that is not yet in this chapter, we would be delighted to have you tell us about it in the form of a usecase. Please see the contributing guide for more info.

Part V

Appendix

GLOSSARY

absolute path The complete path from the root of the file system. Absolute paths always start with /. Example: /home/user/Pictures/xkcd-webcomics/530.png. See also RELATIVE PATH.

adjusted branch git-annex concept: a special BRANCH in a dataset. Adjusted branches refer to a different, existing branch that is not adjusted. The adjusted branch is called "adjusted/<branchname>(unlocked)" and on an the adjusted branch", all files handled by GIT-ANNEX are not locked – They will stay "unlocked" and thus modifiable. Instead of referencing data in the ANNEX with a SYMLINK, unlocked files need to be copies of the data in the annex. Adjusted branches primarily exist as the default branch on so-called CRIPPLED FILESYSTEMS such as Windows.

annex git-annex concept: a different word for OBJECT-TREE.

annex UUID A UUID assigned to an annex of each individual CLONE of a dataset repository. GIT-ANNEX uses this UUID to track file content availability information. The UUID is available under the configuration key annex.uuid and is stored in the configuration file of a local clone (<dataset root>/.git/config). A single dataset instance (i.e. a local clone) has exactly one annex UUID, but other clones of the same dataset each have their own unique annex UUIDs.

bare Git repositories A bare Git repository is a repository that contains the contents of the .git directory of regular DataLad datasets or Git repositories, but no worktree or checkout. This has advantages: The repository is leaner, it is easier for administrators to perform garbage collections, and it is required if you want to push to it at all times. You can find out more on what bare repositories are and how to use them here[679].

bash A Unix SHELL and command language.

Bitbucket Bitbucket is an online platform where one can store and share version controlled projects using Git (and thus also DataLad project), similar to GITHUB or GITLAB. See bitbucket.org[680].

branch Git concept: A lightweight, independent history streak of your dataset. Branches can contain less, more, or changed files compared to other branches, and one can MERGE the changes a branch contains into another branch.

checksum An alternative term to SHASUM.

clone Git concept: A copy of a GIT repository. In Git-terminology, all "installed" datasets are clones.

---

[679] https://git-scm.com/book/en/v2/Git-on-the-Server-Getting-Git-on-a-Server
[680] https://bitbucket.org/

commit Git concept: Adding selected changes of a file or dataset to the repository, and thus making these changes part of the revision history of the repository. Should always have an informative COMMIT MESSAGE.

commit message Git concept: A concise summary of changes you should attach to a `datalad save` command. This summary will show up in your DATALAD DATASET history.

compute node A compute node is an individual computer, part of a HIGH-PERFORMANCE COMPUTING (HPC) or HIGH-THROUGHPUT COMPUTING (HTC) cluster.

conda A package, dependency, and environment management system for a number of programming languages. Find out more at docs.conda.io[681]. It overlaps with PIP in functionality, but it is advised to not use both tools simultaneously for package management.

container recipe A text file template that lists all required components of the computational environment that a SOFTWARE CONTAINER should contain. It is made by a human user.

container image Container images are *built* from CONTAINER RECIPE files. They are a static filesystem inside a file, populated with the software specified in the recipe, and some initial configuration.

crippled filesystem git-annex concept: A file system that does not allow making symlinks or removing write PERMISSIONS from files. Examples for this are FAT[682] (likely used by your USB sticks) or NTFS[683] (used on Windows systems of the last three decades).

DataLad dataset A DataLad dataset is a Git repository that may or may not have a data annex that is used to manage data referenced in a dataset. In practice, most DataLad datasets will come with an annex.

DataLad extension Python packages that equip DataLad with specialized commands. The section *DataLad extensions* (page 285) gives and overview of available extensions and links to Handbook chapters that contain demonstrations.

DataLad subdataset A DataLad dataset contained within a different DataLad dataset (the parent or DATALAD SUPERDATASET).

DataLad superdataset A DataLad dataset that contains one or more levels of other DataLad datasets (DATALAD SUBDATASET).

dataset ID A UUID that identifies a dataset as a unit – across its entire history and flavors. This ID is stored in a dataset's own configuration file (`<dataset root>/.datalad/config`) under the configuration key `datalad.dataset.id`. As this configuration is stored in a file that is part of the Git history of a dataset, this ID is identical for all CLONEs of a dataset and across all its versions.

Debian A common Linux distribution. More information here[684].

debugging Finding and resolving problems within a computer program. To learn about debugging a failed execution of a DataLad command, take a look at the section *Debugging* (page 267).

Docker Docker[685] is a containerization software that can package software into SOFTWARE CONTAINERs, similar to SINGULARITY. Find out more on Wikipedia[686].

---

[681] https://docs.conda.io/en/latest/
[682] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
[683] https://en.wikipedia.org/wiki/NTFS
[684] https://www.debian.org/index.en.html
[685] https://www.docker.com/
[686] https://en.wikipedia.org/wiki/Docker_(software)

Docker-Hub Docker Hub[687] is a library for DOCKER CONTAINER IMAGEs. Among other things, it hosts and builds Docker container images. You can can *pull* CONTAINER IMAGEs built from a publicly shared CONTAINER RECIPE from it.

DOI A digital object identifier (DOI) is a character string used to permanently identify a resource and link to in on the web. A DOI will always refer to the one resource it was assigned to, and only that one.

extractor DataLad concept: A metadata extractor of the DATALAD EXTENSION `datalad-metalad` enables DataLad to extract and aggregate special types of metadata.

environment variable A variable made up of a name/value pair. Programs using a given environment variable will use its associated value for their execution. You can find out a bit more on environment variable *in this Findoutmore* (page 125).

ephemeral clone dataset clones that share the annex with the dataset they were cloned from, without GIT-ANNEX being aware of it. On a technical level, this is achieved via symlinks. They can be created with the `--reckless ephemeral` option of **datalad clone**.

force-push Git concept; Enforcing a **git push** command with the `--force` option. Find out more in the documentation of git push[688].

fork Git concept on repository hosting sites (GitHub, GitLab, Gin, . . .) A fork is a copy of a repository on a web-based Git repository hosting site. Find out more here[689].

GIN A web-based repository store for data management that you can use to host and share datasets. Find out more about GIN here[690].

Git A version control system to track changes made to small-sized files over time. You can find out more about git in this (free) book[691] or these interactive Git tutorials[692] on GITHUB.

git-annex A distributed file synchronization system, enabling sharing and synchronizing collections of large files. It allows managing files with GIT, without checking the file content into Git.

git-annex branch This BRANCH exists in your dataset if the dataset contains an ANNEX. The git-annex branch is completely unconnected to any other branch in your dataset, and contains different types of log files. Its contents are used for git-annex's internal tracking of the dataset and its annexed contents. The branch is managed by GIT-ANNEX, and you should not tamper with it unless you absolutely know what you are doing.

Git config file A file in which GIT stores configuration option. Such a file usually exists on the system, user, and repository (dataset) level.

GitHub GitHub is an online platform where one can store and share version controlled projects using Git (and thus also DataLad project). See'GitHub.com <https://github.com/>'_.

Gitk A repository browser that displays changes in a repository or a selected set of commits. It visualizes a commit graph, information related to each commit, and the files in the trees of each revision.

---

[687] https://hub.docker.com/
[688] https://git-scm.com/docs/git-push#Documentation/git-push.txt---force
[689] https://docs.github.com/en/github/getting-started-with-github/fork-a-repo
[690] https://gin.g-node.org/G-Node/Info/wiki
[691] https://git-scm.com/book/en/v2
[692] https://try.github.io/

GitLab An online platform to host and share software projects version controlled with GIT, similar to GITHUB. See Gitlab.com[693].

globbing A powerful pattern matching function of a shell. Allows to match the names of multiple files or directories. The most basic pattern is `*`, which matches any number of character, such that `ls *.txt` will list all `.txt` files in the current directory. You can read about more about Pattern Matching in Bash's Docs[694].

high-performance computing (HPC) Aggregating computing power from a bond of computers in a way that delivers higher performance than a typical desktop computer in order to solve computing tasks that require high computing power or demand a lot of disk space or memory.

high-throughput computing (HTC) A computing environment build from a bond of computers and tuned to deliver large amounts of computational power to allow parallel processing of independent computational jobs. For more information, see this Wikipedia entry[695].

http Hypertext Transfer Protocol; A protocol for file transfer over a network.

https Hypertext Transfer Protocol Secure; A protocol for file transfer over a network.

logging Automatic protocol creation of software processes, for example in order to gain insights into errors. To learn about logging to troubleshoot problems or remove or increase the amount of information printed to your terminal during the execution of a DataLad command, take a look at the section *Logging* (page 263).

log level Adjusts the amount of verbosity during LOGGING.

Makefile Makefiles are recipes on how to create a digital object for the build automation tool Make[696]. They are used to build programs, but also to manage projects where some files must be automatically updated from others whenever the others change. An example of a Makefile is shown in the usecase *Writing a reproducible paper* (page 399).

manpage Abbreviation of "manual page". For most Unix programs, the command `man <program-name>` will open a PAGER with this commands documentation. If you have installed DataLad as a Debian package, `man` will allow you to open DataLad manpages in your terminal.

master Git concept: For the longest time, `master` was the name of the default BRANCH in a dataset. More recently, the name `main` is used. If you are not sure, you can find out if your default branch is `main` or `master` by running `git branch`.

merge Git concept: to integrate the changes of one BRANCH/SIBLING/ ... into a different branch.

merge request See PULL REQUEST.

metadata "Data about data": Information about one or more aspects of data used to summarize basic information, for example means of create of the data, creator or author, size, or purpose of the data. For example, a digital image may include metadata that describes how large the picture is, the color depth, the image resolution, when the image was created, the shutter speed, and other data.

nano A common text-editor.

---

[693] https://about.gitlab.com/
[694] https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Pattern-Matching
[695] https://en.wikipedia.org/wiki/High-throughput_computing
[696] https://en.wikipedia.org/wiki/Make_(software)

object-tree git-annex concept: The place where GIT-ANNEX stores available file contents. Files that are annexed get a SYMLINK added to GIT that points to the file content. A different word for ANNEX.

Open Science Framework (OSF) An open source software project that facilitates open collaboration in science research.

pager A terminal paper[697] is a program to view file contents in the TERMINAL. Popular examples are the programs less and more. Some terminal output can be opened automatically in a pager, for example the output of a **git log** command. You can use the arrow keys to navigate and scroll in the pager, and the letter q to exit it.

permissions Access rights assigned by most file systems that determine whether a user can view (read permission), change (write permission), or execute (execute permission) a specific content.

- read permissions grant the ability to a file, or the contents (file names) in a directory.

- write permissions grant the ability to modify a file. When content is stored in the OBJECT-TREE by GIT-ANNEX, your previously granted write permission for this content is revoked to prevent accidental modifications.

- execute permissions grant the ability to execute a file. Any script that should be an executable needs to get such permission.

pip A Python package manager. Short for "Pip installs Python". pip install <package name> searches the Python package index PyPi[698] for a package and installs it while resolving any potential dependencies.

provenance A record that describes entities and processes that were involved in producing or influencing a digital resource. It provides a critical foundation for assessing authenticity, enables trust, and allows reproducibility.

publication dependency DataLad concept: An existing SIBLING is linked to a new sibling so that the existing sibling is always published prior to the new sibling. The existing sibling could be a SPECIAL REMOTE to publish file contents stored in the dataset ANNEX automatically with every **datalad push** to the new sibling. Publication dependencies can be set with the option publish-depends in the commands **datalad siblings**, **datalad create-sibling**, and **datalad create-sibling-github/gitlab**.

pull request Also known as MERGE REQUEST. Contributions to Git repositories/DataLad datasets can be proposed to be MERGEd into the dataset by "requesting a pull/update" from the dataset maintainer to obtain a proposed change from a dataset clone or sibling. It is implemented as a feature in repository hosting sites such as GITHUB, GIN, or GITLAB.

relative path A path related to the present working directory. Relative paths never start with /. Example: ../Pictures/xkcd-webcomics/530.png. See also ABSOLUTE PATH.

remote Git-terminology: A repository (and thus also DATALAD DATASET) that a given repository tracks. A SIBLING is DataLad's equivalent to a remote.

Remote Indexed Archive (RIA) store A Remote Indexed Archive (RIA) Store is a flexible and scalable dataset storage solution, useful for collaborative, back-up, or storage workflows. Read more about RIA stores in the section *Remote Indexed Archives for dataset storage and backup* (page 294).

---

[697] https://en.wikipedia.org/wiki/Terminal_pager
[698] https://pypi.org/

run procedure DataLad concept: An executable (such as a script) that can be called with the **datalad run-procedure** command and performs modifications or routine tasks in datasets. Procedures can be written by users, or come with DataLad and its extensions. Find out more in section *Configurations to go* (page 126)

run record A command summary of a **datalad run** command, generated by DataLad and included in the commit message.

sed A Unix stream editor to parse and transform text. Find out more here[699] and in its documentation[700].

shasum A hexadecimal number, 40 digits long, that is produced by a secure hash algorithm, and is used by GIT to identify COMMITs. A shasum is a type of CHECKSUM.

shebang The characters #! at the very top of a script. One can specify the interpreter (i.e., the software that executes a script of yours, such as Python) after with it such as in #! /usr/bin/python. If the script has executable PERMISSIONS, it is henceforth able to call the interpreter itself. Instead of python code/myscript.py one can just run code/myscript if myscript has executable PERMISSIONS and a correctly specified shebang.

shell A command line language and programming language. See also TERMINAL.

special remote git-annex concept: A protocol that defines the underlying transport of annexed files to and from places that are not GIT repositories (e.g., a cloud service or external machines such as HPC systems).

squash Git concept; Squashing is a Git operation which rewrites history by taking a range of commits and squash them into a single commit. For more information on rewriting Git history, checkout section *Back and forth in time* (page 245) and the documentation[701].

SSH Secure shell (SSH) is a network protocol to link one machine (computer), the *client*, to a different local or remote machine, the *server*. See also: SSH SERVER.

SSH key An SSH key is an access credential in the SSH protocol that can be used to login from one system to remote servers and services, such as from your private computer to an SSH SERVER, without supplying your username or password at each visit. To use an SSH key for authentication, you need to generate a key pair on the system you would like to use to access a remote system or service (most likely, your computer). The pair consists of a *private* and a *public* key. The public key is shared with the remote server, and the private key is used to authenticate your machine whenever you want to access the remote server or service. Services such as GITHUB, GITLAB, and GIN use SSH keys and the SSH protocol to ease access to repositories. This tutorial by GitHub[702] is a detailed step-by-step instruction to generate and use SSH keys for authentication.

SSH server An remote or local computer that users can log into using the SSH protocol.

stdin Unix concept: One of the three standard input/output streams[703] in programming. Standard input (stdin) is a stream from which a program reads its input data.

stderr Unix concept: One of the three standard input/output streams[704] in programming. Standard error (stderr) is a stream to which a program outputs error messages, independent

---

[699] https://en.wikipedia.org/wiki/Sed
[700] https://www.gnu.org/software/sed/manual/sed.html
[701] https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History
[702] https://help.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
[703] https://en.wikipedia.org/wiki/Standard_streams
[704] https://en.wikipedia.org/wiki/Standard_streams

from standard output.

stdout Unix concept: One of the three standard input/output streams[705] in programming. Standard output (stdout) is a stream to which a program writes its output data.

symlink A symbolic link (also symlink or soft link) is a reference to another file or path in the form of a relative path. Windows users are familiar with a similar concept: shortcuts.

sibling DataLad concept: A dataset clone that a given DataLad dataset knows about. Changes can be retrieved and pushed between a dataset and its sibling. It is the equivalent of a remote in Git.

Singularity Singularity[706] is a containerization software that can package software into software containers. It is a useful alternative to Docker as it can run on shared computational infrastructure. Find out more on Wikipedia[707].

Singularity-Hub singularity-hub.org[708] is a Singularity container portal. Among other things, it hosts and builds Singularity container images. You can can *pull* container images built from a publicly shared container recipe from it.

software container Computational containers are cut-down virtual machines that allow you to package software libraries and their dependencies in precise versions into a bundle that can be shared with others. They are running instances of a container image. On your own and other's machines, the container constitutes a secluded software environment that contains the exact software environment that you specified but does not effect any software outside of the container. Unlike virtual machines, software containers do not have their own operating system and instead use basic services of the underlying operating system of the computer they run on (in a read-only fashion). This makes them lightweight and portable. By sharing software environments with containers, such as Docker or Singularity containers, others (and also yourself) have easy access to software without the need to modify the software environment of the machine the container runs on.

submodule Git concept: a submodule is a Git repository embedded inside another Git repository. A DataLad subdataset is known as a submodule in the Git config file.

tab completion Also known as command-line completion. A common shell feature in which the program automatically fills in partially types commands upon pressing the TAB key.

tag Git concept: A mark on a commit that can help to identify commits. You can attach a tag with a name of your choice to any commit by supplying the --version-tag <TAG-NAME> option to **datalad save**.

the DataLad superdataset /// DataLad provides unified access to a large amount of data at an open data collection found at datasets.datalad.org[709]. This collection is known as "The DataLad superdataset" and under its shortcut, ///. You can install the superdataset – and subsequently query its content via metadata search – by running datalad clone ///.

tig A text-mode interface for git that allows you to easily browse through your commit history. It is not part of git and needs to be installed. Find out more here[710].

terminal The terminal (sometimes also called a shell, console, or CLI) is an interactive, text based interface that allows you to access your computer's functionality. The most common

---

[705] https://en.wikipedia.org/wiki/Standard_streams
[706] https://sylabs.io/docs/
[707] https://en.wikipedia.org/wiki/Singularity_(software)
[708] https://singularity-hub.org/
[709] http://datasets.datalad.org/
[710] https://jonas.github.io/tig/

command-line shells use BASH or c-shell. You can get a short intro to the terminal and useful commands in the section *General prerequisites* (page 22).

Ubuntu A common Linux distribution. More information here[711].

UUID Universally Unique Identifier. It is a character string used for *unambiguous*, identification, formatted according to a specific standard. This identification is not only unambiguous and unique on a system, but indeed *universally* unique – no UUID exists twice anywhere *on the planet*. Every DataLad dataset has a UUID that identifies a dataset uniquely as a whole across its entire history and flavors called DATASET ID that looks similar to this 0828ac72-f7c8-11e9-917f-a81e84238a11. This dataset ID will only exist once, identifying only one particular dataset on the planet. Note that this does not require all UUIDs to be known in some central database – the fact that no UUID exists twice is achieved by mere probability: The chance of a UUID being duplicated is so close to zero that it is negligible.

version control Processes and tools to keep track of changes to documents or other collections of information.

vim A text editor, often the default in UNIX operating systems. If you are not used to using it, but ended up in it accidentally: press ESC : q ! Enter to exit without saving. Here is help: A vim tutorial[712] and how to configure the default editor for git[713].

virtual environment A specific Python installation with packages of your choice, kept in a self-contained directory tree, and not interfering with the system-wide installations. Virtual environments are an easy solution to create several different Python environments and come in handy if you want to have a cleanly structured software setup and several applications with software requirements that would conflict with each other in a single system: You can have one virtual environment with package A in version X, and a second one with package A in version Y. There are several tools that create virtual environments such as the built-in venv module, the virtualenv module, or CONDA. Virtual environments are light-weight and you can switch between them fast.

WSL The Windows Subsystem for Linux, a compatibility layer for running Linux destributions on recent versions of Windows. Find out more here[714].

zsh A Unix shell.

---

[711] https://ubuntu.com
[712] https://www.openvim.com/
[713] https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration
[714] https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux

FREQUENTLY ASKED QUESTIONS

This section answers frequently asked questions about high-level DataLad concepts or commands. If you have a question you want to see answered in here, please create an issue[715] or a pull request[716]. For a series of specialized command snippets for various use cases, please see section *Gists* (page 273).

B.1 What is Git?

Git is a free and open source distributed version control system. In a directory that is initialized as a Git repository, it can track small-sized files and the modifications done to them. Git thinks of its data like a *series of snapshots* – it basically takes a picture of what all files look like whenever a modification in the repository is saved. It is a powerful and yet small and fast tool with many features such as *branching and merging* for independent development, *checksumming* of contents for integrity, and *easy collaborative workflows* thanks to its distributed nature.

DataLad uses Git underneath the hood. Every DataLad dataset is a Git repository, and you can use any Git command within a DataLad dataset. Based on the configurations in `.gitattributes`, file content can be version controlled by Git or managed by git-annex, based on path pattern, file types, or file size. The section *More on DIY configurations* (page 117) details how these configurations work. This chapter[717] gives a comprehensive overview on what Git is.

B.2 Where is Git's "staging area" in DataLad datasets?

As mentioned in *Populate a dataset* (page 37), a local version control workflow with DataLad "skips" the staging area (that is typical for Git workflows) from the user's point of view.

B.3 What is git-annex?

git-annex (https://git-annex.branchable.com/) is a distributed file synchronization system written by Joey Hess. It can share and synchronize large files independent from a commercial service or a central server. It does so by managing all file *content* in a separate directory (the *annex*, *object tree*, or *key-value-store* in `.git/annex/objects/`), and placing only file names and metadata into version control by Git. Among many other features, git-annex can ensure sufficient amounts of file copies to prevent accidental data loss and enables a variety of data transfer

---

[715] https://github.com/datalad-handbook/book/issues/new
[716] http://handbook.datalad.org/en/latest/contributing.html
[717] https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

mechanisms. DataLad uses git-annex underneath the hood for file content tracking and transport logistics. git-annex offers an astonishing range of functionality that DataLad tries to expose in full. That being said, any DataLad dataset (with the exception of datasets configured to be pure Git repositories) is fully compatible with git-annex – you can use any git-annex command inside a DataLad dataset.

The chapter *Under the hood: git-annex* (page 83) can give you more insights into how git-annex takes care of your data. git-annex's website[718] can give you a complete walk-through and detailed technical background information.

B.4 What does DataLad add to Git and git-annex?

DataLad sits on top of Git and git-annex and tries to integrate and expose their functionality fully. While DataLad thus is a "thin layer" on top of these tools and tries to minimize the use of unique/idiosyncratic functionality, it also tries to simplify working with repositories and adds a range of useful concepts and functions:

- Both Git and git-annex are made to work with a single repository at a time. For example, while nesting pure Git repositories is possible via Git submodules (that DataLad also uses internally), *cleaning up* after placing a random file somewhere into this repository hierarchy can be very painful. A key advantage that DataLad brings to the table is that it makes the boundaries between repositories vanish from a user's point of view. Most core commands have a `--recursive` option that will discover and traverse any subdatasets and do-the-right-thing. Whereas git and git-annex would require the caller to first cd to the target repository, DataLad figures out which repository the given paths belong to and then works within that repository. `datalad save . --recursive` will solve the subdataset problem above for example, no matter what was changed/added, no matter where in a tree of subdatasets.

- DataLad provides users with the ability to act on "virtual" file paths. If software needs data files that are carried in a subdataset (in Git terms: submodule) for a computation or test, a `datalad get` will discover if there are any subdatasets to install at a particular version to eventually provide the file content.

- DataLad adds metadata facilities for metadata extraction in various flavors, and can store extracted and aggregated metadata under `.datalad/metadata`.

-

B.5 Does DataLad host my data?

No, DataLad manages your data, but it does not host it. When publishing a dataset with annexed data, you will need to find a place that the large file content can be stored in – this could be a web server, a cloud service such as Dropbox[719], an S3 bucket, or many other storage solutions – and set up a publication dependency on this location. This gives you all the freedom to decide where your data lives, and who can have access to it. Once this set up is complete, publishing and accessing a published dataset and its data are as easy as if it would lie on your own machine. You can find a typical workflow in the chapter *Third party infrastructure* (page 177).

---

[718] https://git-annex.branchable.com/
[719] https://www.dropbox.com/

B.6 How does GitHub relate to DataLad?

DataLad can make good use of GitHub, if you have figured out storage for your large files otherwise. You can make DataLad publish file content to one location and afterwards automatically push an update to GitHub, such that users can install directly from GitHub and seemingly also obtain large file content from GitHub. GitHub is also capable of resolving submodule/subdataset links to other GitHub repos, which makes for a nice UI.

B.7 Does DataLad scale to large dataset sizes?

In general, yes. The largest dataset managed by DataLad at this point is the Human Connectome Project[720] data, encompassing 80 Terabytes of data in 15 million files, and larger projects (up to 500TB) are currently actively worked on. The chapter *Go big or go home* (page 323) is a guide to "beyond-household-quantity datasets".

B.8 What is the difference between a superdataset, a subdataset, and a dataset?

Conceptually and technically, there is no difference between a dataset, a subdataset, or a superdataset. The only aspect that makes a dataset a sub- or superdataset is whether it is *registered* in another dataset (by means of an entry in the `.gitmodules`, automatically performed upon an appropriate `datalad install -d` or `datalad create -d` command) or contains registered datasets.

B.9 How can I convert/import/transform an existing Git or git-annex repository into a DataLad dataset?

You can transform any existing Git or git-annex repository of yours into a DataLad dataset by running:

```
$ datalad create -f
```

inside of it. Afterwards, you may want to tweak settings in `.gitattributes` according to your needs (see sections *DIY configurations* (page 112) and *More on DIY configurations* (page 117) for additional insights on this). The chapter *Better late than never* (page 353) guides you through transitioning an existing project into DataLad.

B.10 How can I cite DataLad?

Please cite the official paper on DataLad:

Halchenko et al., (2021). DataLad: distributed system for joint management of code, data, and their relationship. Journal of Open Source Software, 6(63), 3262, https://doi.org/10.21105/joss.03262.

---

[720] http://www.humanconnectomeproject.org/

B.11 How can I help others get started with a shared dataset?

If you want to share your dataset with users that are not already familiar with DataLad, it is helpful to include some information on how to interact with DataLad datasets in your dataset's README (or similar) file. Below, we provide a standard text block that you can use (and adapt as you wish) for such purposes. If you do not want to copy-and-paste these snippets yourself, you can run `datalad add-readme` in your dataset, and have it added automatically.

M2.1 Textblock in .rst format:

DataLad datasets and how to use them

This repository is a `DataLad <https://www.datalad.org/>`__ dataset. It provides
fine-grained data access down to the level of individual files, and allows for
tracking future updates. In order to use this repository for data retrieval,
`DataLad <https://www.datalad.org>`_ is required.
It is a free and open source command line tool, available for all
major operating systems, and builds up on Git and `git-annex
<https://git-annex.branchable.com>`__ to allow sharing, synchronizing, and
version controlling collections of large files. You can find information on
how to install DataLad at `handbook.datalad.org/en/latest/intro/installation.html
<http://handbook.datalad.org/en/latest/intro/installation.html>`_.

Get the dataset
^^^^^^^^^^^^^^^

A DataLad dataset can be ``cloned`` by running::

    datalad clone <url>

Once a dataset is cloned, it is a light-weight directory on your local machine.
At this point, it contains only small metadata and information on the
identity of the files in the dataset, but not actual *content* of the
(sometimes large) data files.

Retrieve dataset content
^^^^^^^^^^^^^^^^^^^^^^^^^

After cloning a dataset, you can retrieve file contents by running::

    datalad get <path/to/directory/or/file>

This command will trigger a download of the files, directories, or
subdatasets you have specified.

DataLad datasets can contain other datasets, so called *subdatasets*. If you
clone the top-level dataset, subdatasets do not yet contain metadata and

M2.2 Textblock in markdown format

    datalad get -n <path/to/subdataset>

Afterwards, you can browse the retrieved metadata to find out about
subdataset contents, and retrieve individual files with ``datalad get``. If you
use ``datalad get <path/to/subdataset>``, all contents of the subdataset will
be downloaded at once.

Stay up-to-date
^^^^^^^^^^^^^^^

DataLad datasets can be updated. The command ``datalad update`` will *fetch*
updates and store them on a different branch (by default
``remotes/origin/master``). Running::

    datalad update --merge

will *pull* available updates and integrate them in one go.

Find out what has been done
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

DataLad datasets contain their history in the ``git log``.

```
[![made-with-datalad](https://www.datalad.org/badges/made_with.svg)](https://
↪datalad.org)

## DataLad datasets and how to use them

This repository is a [DataLad](https://www.datalad.org/) dataset. It provides
fine-grained data access down to the level of individual files, and allows for
tracking future updates. In order to use this repository for data retrieval,
[DataLad](https://www.datalad.org/) is required. It is a free and
open source command line tool, available for all major operating
systems, and builds up on Git and [git-annex](https://git-annex.branchable.com/)
to allow sharing, synchronizing, and version controlling collections of
large files. You can find information on how to install DataLad at
[handbook.datalad.org/en/latest/intro/installation.html](http://handbook.datalad.
↪org/en/latest/intro/installation.html).


### Get the dataset

A DataLad dataset can be `cloned` by running


```
datalad clone <url>
```


Once a dataset is cloned, it is a light-weight directory on your local machine.
At this point, it contains only small metadata and information on the
identity of the files in the dataset, but not actual *content* of the
(sometimes large) data files.

Retrieve dataset content
```

> **M2.3 Textblock without formatting**
>
> DataLad datasets and how to use them
> This repository is a DataLad (https://www.datalad.org/) dataset. It provides fine-grained
> data access down to the level of individual files, and allows for tracking future updates.
> In order to use this repository for data retrieval, DataLad (https://www.datalad.org/)
> is required. It is a free and open source command line tool, available for all major
> operating systems, and builds up on Git and git-annex (https://git-annex.branchable.c
> om/) to allow sharing, synchronizing, and version controlling collections of large files.
> You can find information on how to install DataLad at http://handbook.datalad.org/en/
> latest/intro/installation.html.
> Get the dataset
> A DataLad dataset can be "cloned" by running 'datalad clone <url>'. Once a dataset is
> cloned, it is a light-weight directory on your local machine. At this point, it contains only
> small metadata and information on the identity of the files in the dataset, but not actual
> *content* of the (sometimes large) data files.
> Retrieve dataset content
> After cloning a dataset, you can retrieve file contents by running 'datalad get
> <path/to/directory/or/file>'
> This command will trigger a download of the files, directories, or subdatasets you have
> specified.

```
### Stay up to date

DataLad datasets can be updated. The command `datalad update` will
*fetch* updates and store them on a different branch (by default
```

```
remotes/origin/master`). Running
```
Appendix B. Frequently Asked Questions

```
datalad update --merge
```

> 💡 DataLad datasets can contain other datasets, so called "subdatasets". If you clone the top-level dataset, subdatasets do not yet contain metadata and information on the identity of files, but appear to be empty directories. In order to retrieve file availability metadata in subdatasets, run 'datalad get -n <path/to/subdataset>'
>
> Afterwards, you can browse the retrieved metadata to find out about subdataset contents, and retrieve individual files with *datalad get*. If you use 'datalad get <path/to/subdataset>', all contents of the subdataset will be downloaded at once.
>
> Stay up-to-date
>
> DataLad datasets can be updated. The command 'datalad update' will "fetch" updates and store them on a different branch (by default 'remotes/origin/master'). Running 'datalad update –merge' will "pull" available updates and integrate them in one go.
>
> Find out what has been done
>
> DataLad datasets contain their history in the Git log. By running 'git log' (or a tool that displays Git history) in the dataset or on specific files, you can find out what has been done to the dataset or to individual files by whom, and when.
>
> More information
>
> More information on DataLad and how to use it can be found in the DataLad Handbook at http://handbook.datalad.org/en/latest/index.html. The chapter "DataLad datasets" can help you to familiarize yourself with the concept of a dataset.

B.12 What is the difference between DataLad, Git LFS, and Flywheel?

Flywheel[721] is an informatics platform for biomedical research and collaboration.

Git Large File Storage[722] (Git LFS) is a command line tool that extends Git with the ability to manage large files. In that it appears similar to git-annex.

A more elaborate delineation from related solutions can be found in the DataLad developer documentation[723].

B.13 What is the difference between DataLad and DVC?

DVC[724] is a version control system for machine learning projects. We have compared the two tools in a dedicated handbook section, *Reproducible machine learning analyses: DataLad as DVC* (page 358).

---

[721] https://flywheel.io/
[722] https://github.com/git-lfs/git-lfs
[723] http://docs.datalad.org/en/latest/related.html
[724] https://dvc.org/

B.14 DataLad version-controls my large files – great. But how much is saved in total?

B.15 How can I copy data out of a DataLad dataset?

Moving or copying data out of a DataLad dataset is always possible and works in many cases just like in any regular directory. The only caveat exists in the case of annexed data: If file content is managed with git-annex and stored in the OBJECT-TREE, what *appears* to be the file in the dataset is merely a symlink (please read section *Data integrity* (page 85) for details). Moving or copying this symlink will not yield the intended result – instead you will have a broken symlink outside of your dataset.

When using the terminal command cp[733], it is sufficient to use the -L/--dereference option. This will follow symbolic links, and make sure that content gets moved instead of symlinks. Remember that if you are copying some annexed content out of a dataset without unlocking it first, you will only have "read" PERMISSIONS on the files you have just copied. Therefore you can : - either unlock the files before copying them out, - or copy them and then use the command chmod to be able to edit the file.

```
# this will give you 'write' permission on the file
$ chmod +w filename
```

If you are not familiar with how the chmod works (or if you forgot - let's be honest we all google it sometimes), this is a nice tutorial[725] .

With tools other than cp (e.g., graphical file managers), to copy or move annexed content, make sure it is *unlocked* first: After a **datalad unlock** copying and moving contents will work fine. A subsequent **datalad save** in the dataset will annex the content again.

B.16 Is there Python 2 support for DataLad?

No, Python 2 support has been dropped in September 2019[726].

B.17 Is there a graphical user interface for DataLad?

No, DataLad's functionality is available in the command line or via it's Python API.

B.18 How does DataLad interface with OpenNeuro?

OpenNeuro[727] is a free and open platform for sharing MRI, MEG, EEG, iEEG, and ECoG data. It publishes hosted data as DataLad datasets on GitHub. The entire collection can be found

---

[733] The absolutely amazing Midnight Commander[734] mc can also follow symlinks.
[734] https://github.com/MidnightCommander/mc
[725] https://bids.github.io/2015-06-04-berkeley/shell/07-perm.html
[726] https://github.com/datalad/datalad/pull/3629
[727] https://openneuro.org/

at github.com/OpenNeuroDatasets[728]. You can obtain the datasets just as any other DataLad datasets with `datalad clone` or `datalad install`.

There is more info about this in the OpenNeuro Quickstart Guide.

B.19 BIDS validator issues in datasets with missing file content

As outlined in section *Data integrity* (page 85), all unretrieved files in datasets are broken symlinks. This is desired, and not a problem per se, but some tools, among them the BIDS validator[729], can be confused by this. Should you attempt to validate a dataset in which all or some file contents are missing, for example after cloning a dataset or after dropping file contents, the validator may fail to report on the validity of the complete dataset or the specific unretrieved files. If you aim for a complete validation of your dataset, re-do the validation after retrieving all necessary file contents. If you only aim to validate file names and structure, invoke the bids validator with the additional flags `--ignoreNiftiHeaders` and `--ignoreSymlinks`.

B.20 What is the git-annex branch?

If your DataLad dataset contains an annex, there is also a git-annex BRANCH that is created, used, and maintained solely by GIT-ANNEX. It is completely unconnected to any other branches in your dataset, and contains different types of log files. The contents of this branch are used for git-annex internal tracking of the dataset and its annexed contents. For example, git-annex stores information where file content can be retrieved from in a `.log` file for each object, and if the object was obtained from web-sources (e.g., with `datalad download-url`), a `.log.web` file stores the URL. Other files in this branch store information about the known remotes of the dataset and their description, if they have one. You can find out much more about the git-annex branch and its contents in the documentation[730]. This branch, however, is managed by git-annex, and you should not tamper with it.

B.21 Help - Why does Github display my dataset with git-annex as the default branch?

If your dataset is represented on GitHub with cryptic directories instead of actual file names, GitHub probably declared the GIT-ANNEX BRANCH to be your repositories "default branch". Here is an example:

This is related to GitHub's decision to make `main` the default branch for newly created repositories[731] – datasets that do not have a `main` branch (but for example a `master` branch) may end up with a different branch being displayed on GitHub than intended.

To fix this for present and/or future datasets, the default branch can be configured to a branch name of your choice on a repository- or organizational level via GitHub's web-interface[732]. Alternatively, you can rename existing `master` branches into `main` using `git branch -m master main` (but beware of unforeseen consequences - your collaborators may try to update the `master`

---

[728] https://github.com/OpenNeuroDatasets
[729] https://github.com/bids-standard/bids-validator
[730] https://git-annex.branchable.com/internals/
[731] https://github.blog/changelog/2020-10-01-the-default-branch-for-newly-created-repositories-is-now-main/
[732] https://github.blog/changelog/2020-08-26-set-the-default-branch-for-newly-created-repositories/

branch but fail, continuous integration workflows could still try to use master, etc.). Lastly, you can initialize new datasets with main instead of master – either with a global Git configuration[735] for init.defaultBranch, or by using the --initial-branch option of datalad create.

---

[735] See the section *DIY configurations* (page 112) for more info on configurations

SO… WINDOWS… EH?

DataLad and its underlying tools work different on Windows machines. This makes the user experience less fun than on other operating systems – an honest assessment.

Many software tools for research or data science are first written and released for Linux, then for Mac, and eventually Windows. TensorFlow for Windows was released only a full year after it became open source[736], for example, and Python only became easy to install on Windows in 2019[737]. The same is true for DataLad and its underlying tools. There *is* Windows support and user documentation, but it isn't as far developed as for Unix-based systems. This page summarizes core downsides and deficiencies of Windows, DataLad on Windows, and the user documentation.

---

[736] https://developers.googleblog.com/2016/11/tensorflow-0-12-adds-support-for-windows.html
[737] https://devblogs.microsoft.com/python/python-in-the-windows-10-may-2019-update/

C.1 Windows-Deficiencies

Updates

We have succeeded in building git-annex with MagicMime for mimeencoding (that's a good thing). While we're working on packaging everything into a single, conda-based DataLad installation bundle, you can find a standalone git-annex installer at http://datasets.datalad.org/datalad/packages/windows/.

Windows works fundamentally different than macOS or Linux-based operating systems. This results in missing dependencies, altered behavior, and inconvenient workarounds. Beyond this, Windows uses a different file system than Unix based systems. Given that DataLad is a data management software, it is heavily affected by this, and the Basics part of the handbook is filled with "Windows-Wits", dedicated sections that highlight different behavior on native Windows installations of DataLad, or provide adjusted commands – nevertheless, standard DataLad operations on Windows can be much slower than on other operating systems.

A major annoyance and problem is that some tools that DataLad or DATALAD EXTENSIONs use are not available on Windows. If you are interested in adding SOFTWARE CONTAINERs to your DataLad dataset (with the `datalad-container` extension), for example, you will likely not be able to do so on a native Windows computer – SINGULARITY, a widely used containerization software, doesn't exit for Windows, and while there *is* some support for DOCKER on Windows, it does not apply to most private computers[753].

Windows also has insufficient support for SYMLINKing and locking files (i.e., revoking write PERMISSIONS), which alters how GIT-ANNEX works, and may make interoperability of datasets between Windows and non-Windows operating systems not as smooth as between various flavours of Unix-like operating systems.

In addition, Windows has a (default) maximum path length limitation of only 260 characters[738]. However, DataLad (or rather, GIT-ANNEX) relies on file content hashing[739] to ensure file integrity. Usually, the *longer* the *hash* that is created, the more fail-safe it is. For a general idea about the length of hashes, consider that many tools including GIT-ANNEX use SHA256 (a 256 characters long hash) as their default. As git-annex represents files with their content hash as a name, though, a secure 256 character file name is too long for Windows. Datasets thus adjust this default to a 128 character hash[754], but still, if you place a DataLad dataset into a deeply nested directory location, you may run into issues due to hitting the path length limit[756]. You *can*

---

[753] If you are thinking, "Well, why would you use SINGULARITY, DOCKER is available on Windows!": True, and `datalad-container` can indeed use Docker. But Docker can only be installed on Windows Pro or Enterprise, but not on Windows Home. Eh. :(

[738] https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation

[739] https://en.wikipedia.org/wiki/Hash_function

[754] The path length limitation on Windows is the reason that DataLad datasets always use hashes based on MD5[755], a hash function that produces a 128 character hash value. This wouldn't be necessary on Unix-based operating systems, but is required to ensure portability of datasets to Windows computers.

[755] https://en.wikipedia.org/wiki/MD5

[756] The path length limitation certainly isn't only a problem for DataLad and its underlying tools. Many users run into a Path length related problems at least once, by accident. Downloading or copying files with long names into a folder that itself has a long name, for example, can become an unexpected issue (especially if you are not aware of the limit). Imagine transferring pictures from your friends camera into `C:\Users\"Bob McBobface"\Desktop\Pictures\"Vacation Pictures"\2020\Saint-Remy-en-Bouzemont-Saint-Genest-et-Isson\"From Alice and Sasha"\Camera\` – those file names shouldn't be too long to fit in the limit. Likewise, when `git cloning` a GIT repository that was created on a Unix computer and contains very long file names could fail.

enable long paths in recent builds of Windows 10, but it requires some tweaking[740].

Windows also doesn't really come with a decent TERMINAL. It is easy to get a nice and efficient terminal set up on macOS or Linux, it is harder on Windows. For example, its TAB COMPLETION is deemed inefficient by many, it takes some poking and clicking[741] to enable copy-pasting, most standard command line tools are not pre-installed, and many aren't even available or easy to access from the terminal. Usually, Windows users aren't bothered much by this, but DataLad is a command line tool, and with a command line that is difficult to use, command line tools become difficult to use, too. Are you a Windows user and have tips for setting up a decent terminal? Please tell us, we're eager to learn from you[742].

Sadly, even the non-commandline parts of Windows bear inconveniences. Windows' File Explorer does not display common file extensions by default, and some editors (such as notepad) add their own file extensions to files, even when they already have an extension. This can cause confusion.

Unfortunately, issues that affect Windows itself are out of our hands. We can adapt to limitations, but in many cases it is not possible to overcome them. That sucks, and we're really sorry for this. Its not that we pick dependencies that only work on Unix-based systems – we try to use tools that are as cross-platform-compatible as possible, but certain tools, functions, or concepts simply don't (yet) work on Windows:

- As there is no way to install SINGULARITY or DOCKER on regular Windows machines, none of the functionality that the `datalad-container` extension provides can be used.

- As there is insufficient support for symlinking and locking, datasets will have a higher disk usage on Windows machines. Section *Data integrity* (page 85) has the details on this.

- The Windows terminals are much less user friendly, and errors that are thrown on Windows systems are typically much more complex.

- DataLad and its underlying tools are slower on Windows.

C.2 DataLad-on-Windows-Deficiencies

DataLad is developed and predominantly used on Linux-based operating systems. There is a broad suite of unit tests[743] and continuous integration[744] to ensure that functions and commands work under Windows, but given that development and user base is mostly not Windows-based, many bugs that would only surface during complete workflows (as opposed to atomic unit testing) or on machines with specific configurations, versions, or software environments (as opposed to the simplistic and isolated Windows test environments on continuous integration) have not been discovered yet. And a typical Windows user may also use their computer differently than a Linux-based developer imagines computers to be used.

Thus, when using DataLad under Windows it is likely that you encounter bugs. We're trying to prevent this, but it is a normal part of (scientific) software development. What you can do to help us improve your experience is to talk to us at github.com/datalad/datalad[745] about

---

[740] https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation#enable-long-paths-in-windows-10-version-1607-and-later
[741] https://www.howtogeek.com/353200/how-to-enable-copy-and-paste-keyboard-shortcuts-in-windows-10s-bash-shell/
[742] https://github.com/datalad/datalad
[743] https://en.wikipedia.org/wiki/Unit_testing
[744] https://en.wikipedia.org/wiki/Continuous_integration
[745] https://github.com/datalad/datalad

problems or bugs you ran into, about your typical workflows, and the usecases you are trying to achieve.

C.3 User documentation deficiencies

The DataLad Handbook is tested on Debian and predominantly created by Unix users[757]. Being written by many converted Linux users, is filled with start-to-end instructions and tips for Unix systems that have sufficient detail to help Unix newcomers to get started, and it aims to be accessible to everyone – you don't need to be a Linux crank to be able to use the handbook.

However, you may need to be a Windows crank (or a fearless experimentalist) if you want to use all of the handbook on a Windows computer, though. There hasn't been nearly as much time invested into finding, describing, and solving caveats or edge cases, and there isn't enough "daily Windows usage" expertise to be able to give all of the advice that may be needed to identify or prevent problems or improve the user experience to the maximum.

The workflow-based and user-centric narrative of the Basics has been developed on a Unix-based system – Windows-related enhancements are solely adjustments or workarounds. So far, only the *Basics* (page 33) have been tested with a Windows computer (Windows 10, build-version 2004) and adjusted where necessary. We're working on more adjustments, testing, and general improvements, but its a process. You can help us prioritize Windows by getting in touch to voice general interest, discover and report bugs, or contribute to the user documentation with your own advice and experiences.

C.4 So, overall…

You won't get the best possible DataLad experience on a Windows computer. While basic functionality is ensured, it is smoother and more fail-safe to use DataLad on anything but a Native Windows installation, at least for the time being. When sticking to Windows, though, you could find out about interesting aspects of your operating system and help us improve Windows functionality if you tell us about your workflows or report bugs.

C.5 Are there feasible alternatives?

If you want to use DataLad, but fear problems when on Windows, what is there that you can do? Should you switch your operating system?

Its quite easy to say "Just use Linux" but tough to do when you have no experience, support, or spare time and are hence reluctant to completely overhaul your operating system and reduce your productivity while you get a hang on it – or if you rely on software that is native to Windows, such as Microsoft or Adobe products. Its also easy to say "Just use a Mac, its much more user-friendly *and* Unix-based" when an Apple product is a very expensive investment that only few people can or want to afford. Its a bit like recommending a MatLab user (proprietary, expensive, closed-source software) to switch to Python, R, Julia, or a similar open source alternative. Yes, there are real benefits to it that make the change worthwhile to many, but that

[757] Its not written by Windows-lynching ideologists and Linux cranks, though. The lead author switched from Windows to Debian 1.5 years before starting to write the handbook, coming from more than a decade of happy Windows experience. She doesn't regret having made the change at all, but she respects and understands reluctance to switch.

doesn't change the fact that it is effortful and may be frustrating. But how about switching from MatLab to Octave[746], an open source programming language, made to be compatible to Mat-Lab? There definitely is work and adjustment involved, but much less work than when trying to rewrite your analyses in Go or C++. It is tough if you have been using "a thing" for decades without much hassle and now someone tells you to change. If you feel that you lack the time, resources, support, or knowledge, then throwing yourself into cold water and making a harsh change not only sucks, but its also not likely to succeed. If you're juggling studies (or the general publish-or-perish-work-life-misery that Academia too often is), care-giving responsibilities, and surviving a pandemic, all while being in a scientific lab that advocates using Windows and works exclusively with Microsoft Excel, then switching to Arch Linux would widely be seen as a bad idea.

But is there a middle-ground, the "Octave" of switching Operating Systems or alternative solutions? It depends on what you need and what you want to do. Below, we have listed solutions that may be feasible for you as an alternative to native Windows so that you can debate individual pros and cons of each alternative with yourself.

Use a compute cluster

If you are a researcher, chances are that your institution runs a large compute cluster. Those things run on Linux distributions, they have knowledgeable system administrators, and typically institute-internal documentation. Even if you are on a Windows computer, you can log into such a cluster (if you have an account), and use tools made for Unix-like operating systems there – without having to deal with any of the set-up, installation, or maintenance, and with access to documentation and experienced users. The section *Installation and configuration* (page 10) also contains installation instructions for such shared compute clusters ("Linux machines with no root access").

The Windows Subsystem for Linux (version 2)

If you want to have a taste of Unix on your own computer, but in the most safe and reversible way, or have essential software that only runs under Windows and really need to keep a Windows Operating System, then the Windows Subsystem for Linux (WSL2) may be a solution. Microsoft acknowledges that a lot of software is assuming that the environment in which they run behaves like Linux, and has added a real Linux kernel to Windows with the WSL2[747]. If you enable WSL2 on your Windows 10 computer, you have access to a variety of Linux distributions in the Microsoft store, and you can install them with a single click. The Linux distribution(s) of your choice becomes an icon on your task bar, and you can run windows and Linux in parallel.

What should you be mindful of? WSL is a minimalist tool in that it is made to run BASH or core Linux commands. It does not support graphical user (GUI) interfaces or applications. So while common Linux distributions have GUIs for various software, in WSL2 you will only be able to use a terminal. Also, it is important to know that older versions of WSL did not allow accessing or modifying Linux files via Windows[748]. Recent versions (starting with Windows 10 version 1903) allow accessing Linux files with Windows tools[749], although some tweaking, explained in *Cross-OS filesharing with symlinks (WSL2 only)* (page 91), is necessary.

---

[746] https://www.gnu.org/software/octave/
[747] https://docs.microsoft.com/en-us/windows/wsl/faq
[748] https://devblogs.microsoft.com/commandline/do-not-change-linux-files-using-windows-apps-and-tools/
[749] https://devblogs.microsoft.com/commandline/whats-new-for-wsl-in-windows-10-version-1903/

How do I start? Microsoft has detailed installation instructions here[750].

Linux Mint

There isn't much that holds you to Windows? The software you use is either already open source or available on Linux or easily replaceable by available alternatives (e.g., libre office instead of Microsoft Word, the Spotify player in a web browser instead of as an App)? But you're reluctant to undergo huge changes when switching operating systems? Then Linux Mint may be a good starting point. Its user interface is not identical to Windows, but also not that far away, it is a mature operating system, its very user-friendly, there is a helpful and welcoming community behind it, and – like all Linux distributions – it is free.

What should I be mindful of? If you're changing your operating system, **create a backup** of your data (unless you do it on a new computer of course). You can't install a new OS and have all data where you left it – pull it onto an external drive, and copy it back to your new OS later. Also, take a couple of minutes and google whether the hardware of your computer is compatible with Linux. Go to your system's settings and find out the name and version of your computer, your graphics card and CPU, and put all of it into a Google search that starts with "Install Linux on <hardware specifications>". Some hardware may need additional configuration or be incompatible with Linux, and you would want to know about this upfront. And don't be afraid to ask or look for help. The internet is a large place and filled with helpful posts and people. Take a look at user forums such as forums.linuxmint.com/[751] – they likely contain the answers to the questions you may have.

How do I start? A nice and comprehensive overview is detailed in this article[752].

---

[750] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[751] https://forums.linuxmint.com/
[752] https://uk.pcmag.com/adobe-photoshop-cc/124238/how-to-make-the-switch-from-windows-to-linux

---

DATALAD CHEAT SHEET

Fig. D.1: A high-resolution version of this cheatsheet is available for download at https://github.com/datalad-handbook/artwork/raw/master/src/datalad-cheatsheet.pdf

CONTRIBUTING

Thanks for being curious about contributing! We greatly appreciate and welcome contributions to this book, be it in the form of an issue[758], quick feedback on DataLad's usability[759], a pull request, or a discussion you had with anyone on the team via a non-GitHub communication channel! To find out how we acknowledge contributions, please read the paragraph *Acknowledging Contributors* (page 504) at the bottom of this page.

If you are considering doing a pull request: Great! Every contribution is valuable, from fixing typos to writing full chapters. The steps below outline how the book "works". It is recommended to also create an issue to discuss changes or additions you plan to make in advance.

E.1 Software setup

Depending on the size of your contribution, you may want to be able to build the book locally to test and preview your changes. If you are fixing typos, tweak the language, or rewrite a paragraph or two, this should not be necessary, and you can safely skip this paragraph and instead take a look into the paragraph *Easy pull requests* (page 501). If you want to be able to build the book locally, though, please follow these instructions:

- datalad install the repository recursively. This ensures that dependent subdatasets are installed as well

```
$ datalad install -r https://github.com/datalad-handbook/book.git
```

- optional, but recommended: Create a virtual environment

---

[758] https://github.com/datalad-handbook/book/issues/new
[759] https://forms.gle/FkNEc7HVaZU5RTYP6

```
$ virtualenv --python=python3 ~/env/handbook
$ . ~/env/handbook/bin/activate
```

- install the requirements and a custom Python helper for the handbook

```
# navigate into the installed dataset
$ cd book
# install required software
$ pip install -r requirements.txt
$ pip install -e .
```

- install `librsvg2-bin` (a tool to render `.svgs`) with your package manager

```
$ sudo apt-get install librsvg2-bin
```

The code examples that need to be executed to build the book (see also the paragraph "Code" in *Directives and demos* (page 499) to learn more about this) are executed inside of the directory `/home/me`. This means that *this directory needs to exist* on your machine. Essentially, `/home/me` is a mock directory set up in order to have identical paths in code snippets regardless of the machine the book is build on: Else, code snippets created on one machine might have the path `/home/adina`, and others created on a second machine `/home/mih`, for example, leading to some potential confusion for readers. Therefore, you need to create this directory, and also – for consistency in the Git logs as well – a separate, mock Git identity (we chose Elena Piscopia[760], the first woman to receive a PhD. Do not worry, this does not mess with your own Git identity):

```
$ sudo mkdir /home/me
$ sudo chown $USER:$USER /home/me
$ HOME=/home/me git config --global user.name "Elena Piscopia"
$ HOME=/home/me git config --global user.email "elena@example.net"
```

Once this is configured, you can build the book locally by running `make build` in the root of the repository, and open it in your browser, for example with `firefox docs/_build/html/index.html`.

In case you need to remove the build files, you can just run `make clean-build`.

E.2 Directives and demos

If you are writing larger sections that contain code, `gitusernotes`, `findoutmores`, or other special directives, please make sure that you read this paragraph.

The book is build with a number of custom directives. If applicable, please use them in the same way they are used throughout the book.

Code: For code that runs inside a dataset such as `DataLad-101`, working directories exist inside of `/home/me`. The `DataLad-101` dataset for example lives in `/home/me/dl-101`. This comes with the advantage that code is tested immediately – if the code snippet contains an error, this error will be written into the book, and thus prevent faulty commands from being published. Running code in a working directory will furthermore build up on the existing history of this dataset, which is very useful if some code relies on working with previously created

---

[760] https://en.wikipedia.org/wiki/Elena_Cornaro_Piscopia

content or dataset history. Build code snippets that add to these working directories by using the `runrecord` directive. Commands wrapped in these will write the output of a command into example files stored inside of the DataLad Handbook repository clone in `docs/PART/_examples` (where `PART` is basics, beyond_basics, or usecases). Make sure to name these files according to the following schema, because they are executed *sequentially*: `_examples/DL-101-1<nr-of-section>-1<nr-of-example>`, e.g., `_examples/DL-101-101-101` for the first example in the first section of the given part. Here is how a `runrecord` directive can look like:

```
.. runrecord:: _examples/DL-101-101-101   # give the path to the resulting file, start
→with _examples
   :language: console
   :workdir: dl-101/DataLad-101    # specify a working directory here. This translates to
→/home/me/dl-101/DataLad-101

   # this is a comment
   $ this line will be executed
```

Afterwards, the resulting example files need to be committed into Git. To clear existing examples in `docs/PART/_examples` and the mock directories in `/home/me`, run `make clean` (to remove working directories and examples for all parts of the book) or `make clean-examples` (to remove only examples and workdirs of the Basics part).

However, for simple code snippets outside of the narrative of `DataLad-101`, simple `code-block::` directives are sufficient.

Other custom directives: Other custom directives are `gitusernote` (for additional Git-related information for Git-users), and `findoutmore` (foldable sections that contain content that goes beyond the basics). Make use of them, if applicable to your contribution.

Creating live code demos out of runrecord directives: The book has the capability to turn code snippets into a script that the tool cast_live[761] can use to cast and execute it in a demonstration shell. This feature is intended for educational courses and other types of demonstrations. The following prerequisites exist:

- A snippet only gets added to a cast, if the `:cast:` option in the `runrecord` specifies a filename where to save the demo to (it does not need to be an existing file).

- If `:realcommand:` options are specified, they will become the executable part of the cast. If note, the code snippet in the code-block of the `runrecord` will become the executable part of the cast.

- An optional `:notes:` lets you add "speakernotes" for the cast.

- Demos are produced upon `make`, but only if the environment variable `CAST_DIR` is set. This should be a path that points to any directory in which demos should be created and saved. An invocation could look like this:

  ```
  $ CAST_DIR=/home/me/casts make
  ```

This is a fully specified `runrecord`:

```
.. runrecord:: _examples/DL-101-101-101
   :language: console
   :workdir: dl-101/DataLad-101
   :cast: dataset_basics   # name of the cast file (will be created/extended in CAST_DIR)
```
<div align="right">(continues on next page)</div>

---
[761] https://github.com/datalad/datalad/blob/master/tools/cast_live

```
   :notes: This is an optional speaker note only visible to presenter during the cast

   # this is a comment and will be written to the cast
   $ this line will be executed and written to the cast
```

IMPORTANT! Code snippets will be turned into casts in the order of execution of runrecords. If you are adding code into an existing cast, i.e., in between two snippets that get written to the same cast, make sure that the cast will still run smoothly afterwards!

Running live code demos from created casts: If you have created a cast, you can use the tool live_cast in tools/ in the DataLad Course[762] to execute them:

```
~ course$ tools/cast_live path/to/casts
```

The section *Teaching with the DataLad Handbook* (page 505) outlines more on this and other teaching materials the handbook provides.

E.3 Easy pull requests

The easiest way to do a pull request is within the web-interface that GitHub and readthedocs[763] provide. If you visit the rendered version of the handbook at handbook.datalad.org[764] and click on the small, floating v:latest element at the lower right-hand side, the Edit option will take you straight to an editor that lets you make your changes and submit a pull request.

Fig. E.1: You can find an easy way to submit a pull request right from within the handbook.

But you of course are also welcome to submit a pull request with whichever other workflow suites you best.

E.4 Desired structure of the book

The book consists of four major parts: Introduction, Basics, Beyond Basics, and Use Cases, plus an appendix. Purpose and desired content of these parts are outlined below. When contributing to one of these sections, please make sure that your contribution stays in the scope of the respective section.

---

[762] https://github.com/datalad-handbook/course
[763] https://readthedocs.org
[764] http://handbook.datalad.org/

Introduction

- An introduction to DataLad, and the problems it aims to be a solution for.

- This part is practically free of hands-on content, i.e., no instructions, no demos. Instead, it is about concepts, analogies, general problems.

- In order to avoid too much of a mental split between a reader's desire to learn how to actually do things vs. conceptual information, the introduction is purposefully kept short and serves as a narrated table of contents with plenty of references to other parts of the book.

Basics

- This part contains hands-on-style content on skills that are crucial for using DataLad productively. It aims to be a continuous tutorial after which readers are able to perform the following tasks:

  - Create and populate own datasets from scratch

  - Consume existing datasets

  - Share datasets on shared an third party infrastructure and collaborate

  - Execute commands or scripts (computationally) reproducible

  - Configure datasets or DataLad operations as needed

  - Use DataLad's metadata capabilities

- The order of topics in this part is determined by the order in which they become relevant for a novice DataLad user.

- Content should be written in a way that explicitly encourages executing the shown commands, up to simple challenges (such as: "find out who the author of the first commit in the installed subdataset XY is").

Beyond Basics

- This part goes beyond the Basics and is a place for documenting advanced or special purpose commands or workflows. Examples for this sections are: Introductions to special-purpose extensions, hands-on technical documentation such as "how to write your own DataLad extension", or rarely encountered use cases for DataLad, such as datasets for large-scale projects.

- This section contains chapters that are disconnected from each other, and not related to any narrative. Readers are encouraged to read chapters or sections that fit their needs in whichever order they prefer.

- Care should be taken to not turn content that could be a usecase into an advanced chapter.

Use Cases

- Topics that do not fit into the introduction or basics parts, but are DataLad-centric, go into this part. Ideal content are concrete examples of how DataLad's concepts and building

blocks can be combined to implement a solution to a problem.

- Any chapter is written as a more-or-less self-contained document that can make frequent references to introduction and basics, but only few, and more general ones to other use cases. This should help with long-term maintenance of the content, as the specifics of how to approach a particular use case optimally may evolve over time, and cross-references to specific functionality might become invalid.

- There is no inherent order in this part, but chapters may be grouped by domain, skill-level, or DataLad functionality involved (or combinations of those).

- Any content in this part can deviate from the examples and narrative used for introduction and basics whenever necessary (e.g., concrete domain specific use cases). However, if possible, common example datasets, names, terms should be adopted, and the broadest feasible target audience should be assumed. Such more generic content should form the early chapters in this part.

- Unless there is reason to deviate, the following structure should be adopted:

  1. Summary/Abstract (no dedicated heading)

  2. *The Challenge*: description what problem will be solved, or which conditions are present when DataLad is not used

  3. *The DataLad Approach*: high-level description how DataLad can be used to address the problem at hand.

  4. *Step-by-Step*: More detailed illustration on how the "DataLad approach" can be implemented, ideally with concrete code examples.

Intersphinx mapping

The handbook tries to provide stable references to commands, concepts, and use cases for Intersphinx Mappings[765]. This can help to robust-ify links – instead of long URLs that are dependent on file or section titles, or references to numbered sections (both can break easily), intersphinx references are meant to stick to contents and reliably point to it via a mapping in the index[766] under `Symbols`. An example intersphinx mapping is done in DataLad[767].

The references take the following shape: `.. _1-001:`

The leading integer indicates the category of reference:

```
1: Command references
2: Concept references
3: Usecase references
```

The later integers are consecutively numbered in order of creation. If you want to create a new reference, just create a reference one integer higher than the previously highest. The currently existing intersphinx references are:

- 1-001: *DataLad cheat sheet* (page 496)

- 1-002: *DataLad, Run!* (page 59)

- 2-001: *YODA: Best practices for data analyses in a dataset* (page 136)

---

[765] https://www.sphinx-doc.org/en/master/usage/extensions/intersphinx.html
[766] http://handbook.datalad.org/en/latest/genindex.html
[767] https://github.com/datalad/datalad/pull/4046

- 2-002: *Data integrity* (page 85)
- 2-003: *DataLad's result hooks* (page 287)
- 3-001: *Building a scalable data storage for scientific computing* (page 443)

E.5 Tweaking the CSS of the book

The custom CSS of the book is controlled by the file `docs/_static/custom.css`. If you have build the book locally by running *make build*, you can directly tweak the custom CSS file in `docs/_build/html/_static/custom.css` to view the changes without having to rebuild the book. But once you have found the proper CSS style you are happy with make sure to save and commit those changes in `docs/_static/custom.css`

E.6 Acknowledging Contributors

If you have helped this project, we would like to acknowledge your contribution in the GitHub repository[768] in our README with allcontributors.org[769], and the project's .zenodo[770] (you can add yourself as second-to-last, i.e. just above Michael) and CONTRIBUTORS.md[771] files. The allcontributors bot[772] will give credit for various types of contributions[773]. We may ask you to open a PR to add yourself to all of our contributing acknowledgements or do it ourselves and let you know.

---

[768] https://github.com/datalad-handbook/book
[769] https://allcontributors.org/
[770] https://github.com/datalad-handbook/book/blob/master/.zenodo.json
[771] https://github.com/datalad-handbook/book/blob/master/CONTRIBUTORS.md
[772] https://github.com/all-contributors
[773] https://allcontributors.org/docs/en/emoji-key

TEACHING WITH THE DATALAD HANDBOOK

The handbook is a free and open source educational instrument made available under a Creative Commons Attribution-ShareAlike (CC-BY-SA) license[780]. We are happy if the handbook serves as a helpful tool for other trainers, and try to provide many useful additional teaching-related functions and contents. Below, you can find them listed:

F.1 Use the handbook as a textbook/syllabus

The Basics sections of the handbook is a stand-alone course that you can refer trainees to. Regardless of background, users should be able to work through this part of the book on their own. From our own teaching experiences, it is feasible and useful to work through any individual basics chapter in one go, and assign them as weekly or bi-weekly readings.

F.2 Use slides from the DataLad course

In parallel to the handbook, we are conducting data management workshops with attendees of every career stage (MSc students up to PIs). The sessions are either part of a lecture series (with bi-weekly 90 minute sessions) or workshops of different lengths. Sessions in the lecture series are based on each chapter. Longer workshops combine several chapters. You can find the slides for the workshops in the companion course repository[774]. Slides are made using reveal.js[775]. They are available as PDFs in talks/PDFs/, or as the source html files in talks/.

---

[780] CC-BY-SA means that you are free to

- share - copy and redistribute the material in any medium or format
- adapt - remix, transform, and build upon the material for any purpose, even commercially

under the following terms:

1. Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
2. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

[774] https://github.com/datalad-handbook/course
[775] https://github.com/hakimel/reveal.js/

F.3 Enhance talks and workshops with code demos

Any number of code snippets in the handbook that are created with the `runrecord` directive can be aggregated into a series of commands that can be sequentially executed as a code demo using the cast_live[776] tool provided in the companion course repository[777]. These code demos allow you to remote-control a second terminal that executes the code snippets upon pressing `Enter` and can provide you with simultaneous speaker notes.

A number of demos exist that accompany the slides for the data management sessions in `casts`, but you can also create your own. To find out how to do this, please consult the section directives and demos[778] in the contributing guide. To use the tool, download the `cast_live` script and the `cast_bash.rc` file that accompanies it (e.g., by simply cloning/installing the course repository), and provide a path to the demo you want to run:

```
$ cast_live casts/01_dataset_basics
```

For existing code demos, the chapter Code from chapters contains numbered lists of code snippets to allow your audience to copy-paste what you execute to follow along.

F.4 Use artwork used in the handbook

The handbook's artwork[779] repository contains the sources for figures used in the handbook.

F.5 Use the handbook as a template for your own teaching material

If you want to document a different software tool in a similar way the handbook does it, please feel free to use the handbook as a template.

---

[776] https://github.com/datalad-handbook/course/blob/master/tools/cast_live
[777] https://github.com/datalad-handbook/course
[778] http://handbook.datalad.org/en/latest/contributing.html#directives-and-demos
[779] https://github.com/datalad-handbook/artwork

ACKNOWLEDGEMENTS

---

[781] http://haxbylab.dartmouth.edu/ppl/yarik.html

[782] https://www.psychoinformatics.de/

[783] https://www.nsf.gov/awardsearch/showAward?AWD_ID=1429999

[784] https://www.gesundheitsforschung-bmbf.de/de/datagit-kombination-von-katalogen-datenbanken-und-verteilungslogistik-in-eine-daten-5607.php

[785] http://cbbs.eu/en/

[786] https://projectreporter.nih.gov/project_info_description.cfm?projectnumber=1P41EB019936-01A1

BOXES, FIGURES, TABLES

H.1 List of important notes

H.2 List of notes for Git users

H.3 List of info boxes

H.4 List of Windows-wits

LIST OF FIGURES