The DataLad Handbook

Release 0.12.0+1.gc4ac88e

Adina S. Wagner 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
Michael Hanke

May 18, 2020

WHAT DATALAD AND THE HANDBOOK ARE ALL ABOUT

Important: PLEASE NOTE: This is an archived version of the DataLad handbook corresponding to its **0.12.0 release** (January 2020), which in turn was corresponding to the 0.12.0 release of DataLad. This handbook version is **not** a complete documentation of all functionality in DataLad 0.12.0, but the state the handbook was in at this time. Find the latest released version of the handbook at handbook.datalad.org/en/stable[1], and its most recent version (including general fixes, visual improvements, and additions of existing commands or workflows based on existing functionality) at handbook.datalad.org/en/latest[2]. The CHANGELOG[3] summarizes the contents and additions that happened between Handbook versions.

[1] http://handbook.datalad.org/en/stable/
[2] http://handbook.datalad.org/en/latest/
[3] https://github.com/datalad-handbook/book/blob/master/CHANGELOG.md

Part I

Welcome to the DataLad handbook!

This handbook is a living resource about why and – more importantly – *how* to use DataLad. It aims to provide novices and advanced users of all backgrounds with both the basics of DataLad and start-to-end use cases of specific applications. If you want to get hands-on experience and learn DataLad, the *Basics* part of this book will teach you. If you want to know what is possible, the *use cases* will show you. And if you want to help others to get started with DataLad, the companion repository[4] provides free and open source teaching material tailored to the handbook.

Before you read on, please note that the handbook is based on **DataLad version 0.12**, but the section *Installation and configuration* (page 19) will set you up with what you need if you currently do not have DataLad 0.12 or higher installed. If you're new here, please start the handbook here.

Important: he handbook is currently in beta stage. If you would be willing to provide feedback on its contents, please get in touch[5].

[5] https://github.com/datalad-handbook/book/issues/new

[4] https://github.com/datalad-handbook/course

Part II

Introduction

CHAPTER

ONE

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 (meta-)data easy**, but it assists with the combination of all things necessary in the digitial 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[6]: 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[7] 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[8], organize your private music library, keep track of all cat memes[9] on the internet, or anything else[10].

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

[6] https://www.go-fair.org/
[7] https://www.go-fair.org/
[8] https://github.com/datalad-datasets/bmmr-t1w-250um
[9] https://www.diabloii.net/gallery/data/500/medium/moar6-cat.jpg
[10] https://media.giphy.com/media/3o6YfXCehdioMXYbcs/giphy.gif

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.

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 DataLad 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[11], Figshare[12] 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 35) 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.

[11] https://www.dropbox.com
[12] https://figshare.com/

CHAPTER

TWO

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 computers 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**.

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
```

(continues on next page)

(continued from previous page)

```
[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.

The book is split in two 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 chapter *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 357) 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 hide them in foldable book sections 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: Click here to show/hide further commands

Sections like this contain content that goes beyond the basics necessary to complete a challenge.

Note further that. . .

Note for Git users:

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.

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[13], write[14] or request[15] an additional chapter if you believe it is a worthwhile addition, or ask a question on Neurostars.org[16] with a **`datalad`** tag if you need help.

[13] http://docs.datalad.org
[14] http://handbook.datalad.org/en/latest/contributing.html
[15] https://github.com/datalad-handbook/book/issues/new
[16] https://neurostars.org/latest

2.2.1 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[17] 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 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.

2.2.2 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`

[17] https://www.go-fair.org/fair-principles/

and `creating` several *DataLad subdataset*s, adding files and changing their content, and executing simple scripts with input data to create results we can share and publish with DataLad.

Find out more: 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[18]. 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.

[18] https://github.com/datalad-handbook/DataLad-101

2.3 Let's get going!

If you have DataLad installed, you can dive straight into chapter 1, *Create a dataset* (page 45). For everyone new, there are the sections *General prerequisites* (page 29) as a minimal tutorial to using the shell and *Installation and configuration* (page 19) to get your DataLad installation set up.

CHAPTER THREE

INSTALLATION AND CONFIGURATION

Note: The handbook is written for DataLad version 0.12. If you already have DataLad installed but are unsure whether it is the correct version, you can get information on your version of DataLad by typing `datalad --version` into your terminal.

3.1 Install DataLad

The content in this chapter is largely based on the information given on the DataLad website[19] and the DataLad documentation[20].

Beyond DataLad itself, the installation requires Python, *Git*, *git-annex*, and potentially Pythons package manager `pip`. The instructions below detail how to install each of these components for different common operating systems. Please file an issue[21] if you encounter problems.

Note that while these installation instructions will provide you with the core DataLad tool, many extensions[22] exist, and they need to be installed separately, if needed.

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

For Debian-based operating systems, the most convenient installation method is to enable the NeuroDebian[23] repository. If you are on a Debian-based system, but do not have the NeuroDebian repository enabled, you should very much consider enabling it right now. The above hyperlink links to a very easy instruction, and it only requires copy-pasting three lines of code. Also, should you be confused by the name: enabling this repository will not do any harm if your field is not neuroscience.

The following command installs DataLad and all of its software dependencies (including the git-annex-standalone package):

[19] https://www.datalad.org/get_datalad.html
[20] http://docs.datalad.org/en/latest/gettingstarted.html
[21] https://github.com/datalad-handbook/book/issues/new
[22] http://docs.datalad.org/en/latest/index.html#extension-packages
[23] http://neuro.debian.net/

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

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

If you want to install DataLad on a machine you do not have root access to, DataLad can be installed with Miniconda[24].

```
$ 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
```

This should install *Git*, *git-annex*, and DataLad. The installer automatically configures the shell to make conda-installed tools accessible, so no further configuration is necessary.

3.1.3 macOS/OSX

A common way to install packages on OS X is via the homebrew[25] package manager. First, install the homebrew package manager. Note that prior to the installation, Xcode[26] needs to be installed from the Mac App Store. Homebrew then can be installed using the command following the instructions on their webpage (linked above).

Next, install git-annex[27]. The easiest way to do this is via `brew`:

```
$ brew install git-annex
```

[24] https://docs.conda.io/en/latest/miniconda.html
[25] https://brew.sh/
[26] https://apps.apple.com/us/app/xcode/id497799835
[27] https://git-annex.branchable.com/install/OSX/

Once git-annex is available, DataLad can be installed via Pythons package manager `pip` as described below. `pip` should already be installed by default. Recent macOS versions may have `pip3` instead of `pip` – use *tab completion* to find out which is installed. If it is `pip3`, run:

```
$ pip3 install datalad~=0.12
```

instead of the code snippets in the section below.

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

```
$ pip3 install --user datalad~=0.12
```

Find out more: If something is not on PATH. . .

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/awagner/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/awagner/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/awagner/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/awagner/Library/Python/3.7/bin
```

3.1.4 Using Pythons package manager pip

DataLad can be installed via Pythons package manager pip[28]. pip comes with Python distributions, e.g., the Python distributions downloaded from python.org[29]. When downloading Python, make sure to chose a recent Python **3** distribution.

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

```
$ pip install datalad~=0.12
```

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

```
$ pip install --user datalad~=0.12
```

In addition, it is necessary to have a current version of *git-annex* installed which is not set up automatically by using the pip method. You can find detailed installation instructions on how to do this here[30].

For Windows, extract the provided EXE installer into an existing Git installation directory (e.g. C:\\Program Files\Git). If done this way, no PATH variable manipulation is necessary.

3.1.5 Windows 10

There are two ways to get DataLad on Windows 10: one is within Windows itself, the other is using WSL, the Windows Subsystem for Linux.

Note: Using Windows itself comes with some downsides. In general, DataLad can feel a bit sluggish on Windows systems. This is because of a range of 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 the 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. If you are a Windows user and want to help improve the handbook for Windows users, please get in touch[31].

1) Install within Windows [RECOMMENDED]

Note: This installation method will get you a working version of DataLad, but be aware that many Unix commands shown in the book examples will not work for you, and DataLad-related output might look different from what we can show in this book. Please get in touch[32] touch if you want to help.

- **Step 1**: Install Conda
  - Go to https://docs.conda.io/en/latest/miniconda.html and pick the latest Python 3 installer. Miniconda is a free, minimal installer for conda and will install conda[33], Python,

[28] https://pip.pypa.io/en/stable/
[29] https://www.python.org
[30] https://git-annex.branchable.com/install/
[31] https://github.com/datalad-handbook/book/issues/new
[32] https://github.com/datalad-handbook/book/issues/new
[33] https://docs.conda.io/en/latest/

depending packages, and a number of useful packages such as pip[34].

- During installation, keep everything on default. In particular, do not add anything to PATH.
- From now on, any further action must take place in the Anaconda prompt, a preconfigured terminal shell. Find it by searching for "Anaconda prompt" in your search bar.

- **Step 2**: Install Git
  - In the Anaconda prompt, run:

    ```
    conda install -c conda-forge git
    ```

    Note: Is has to be from conda-forge, the anaconda version does not provide the cp command.

- **Step 3**: Install git-annex
  - Obtain the current git-annex versions installer from here[35]. Save the file, and double click the downloaded **git-annex-installer.exe** in your Downloads.
  - During installation, you will be prompted to "Choose Install Location". **Install it into the miniconda Library directory**, e.g. C:\Users\me\Miniconda3\Library.

- **Step 4**: Install DataLad via pip
  - pip was installed by miniconda. In the Anaconda prompt, run:

    ```
    pip install datalad~=0.12
    ```

2) Install within WSL

The Windows Subsystem for Linux (WSL) allows Windows users to have full access to a Linux distribution within Windows. If you have always used Windows be prepared for some user experience changes when using Linux compared to Windows. For one, there will be no graphical user interface (GUI). Instead, you will work inside a terminal window. This however mirrors the examples and code snippets provided in this handbook exactly. Using a proper Linux installation improves the DataLad handbook experience on Windows *greatly*. However, it comes with the downside of two filesystems that are somewhat separated. Data access to files within Linux from within Windows is problematic: Note that there will be incompatibilities between the Windows and Linux filesystems. Files that are created within the WSL for example can not be modified with Windows tools. A great resource to get started and understand the WSL is this guide[36].

Requirements:

WSL can be enabled for **64-bit** versions of **Windows 10** systems running **Version 1607** or above. To check whether your computer fulfills these requirements, open *Settings* (in the start menu) > *System* > *About*. If your version number is less than 1607, you will need to perform a windows update[37] before installing WSL.

[34] https://pip.pypa.io/en/stable/
[35] https://downloads.kitenet.net/git-annex/windows/current/
[36] https://github.com/michaeltreat/Windows-Subsystem-For-Linux-Setup-Guide/
[37] https://support.microsoft.com/en-us/help/4028685/windows-10-get-the-update

The instructions below show you how to set up the WSL and configure it to use DataLad and its dependencies. They follow the Microsoft Documentation on the Windows Subsystem for Linux[38]. If you run into troubles during the installation, please consult the WSL troubleshooting page[39].

- **Step 1**: Enable the windows subsystem for Linux
  - Open Windows Power Shell as an Administrator and run

```
$ Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-
↪Subsystem-Linux
```

  - Afterwards, when prompted in the Power Shell, restart your computer
- **Step 2**: Install a Debian Linux distribution
  - To do this, visit the Microsoft store, and search for the Debian distro. We **strongly** recommend installing *Debian*, even though other distributions are available. "Get" the app, and "install" it.
- **Step 3**: Initialize the distribution
  - Launch the Subsystem either from the Microsoft store or from the Start menu. This will start a terminal. Do not worry – there is a dedicated section (*General prerequisites* (page 29)) on how to work with the terminal if you have not so far.
  - Upon first start, you will be prompted to enter a new UNIX username and password. Tip: chose a short name, and no spaces or special characters. The password will become necessary when you elevate a process using sudo – sudo let's you execute a process with rights of another user, such as administrative rights, for examples when you need to install software.
  - Right after initial installation, your Linux distribution will be minimally equipped. Update your package catalog and upgrade your installed packages by running the command below. As with all code examples in this book, make sure to copy commands exactly, including capitalization. If this is the first time you use `sudo`, your system will warn you to use it with care. During upgrading installed packages, the terminal will ask you to confirm upgrades by pressing `Enter`.

```
$ sudo apt update && sudo apt upgrade
```

- **Step 4**: Enable NeuroDebian
  - In your terminal, run

```
$ wget -O- http://neuro.debian.net/lists/stretch.de-md.libre | sudo tee /etc/
↪apt/sources.list.d/neurodebian.sources.list
```

  - Afterwards, run

[38] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[39] https://docs.microsoft.com/en-us/windows/wsl/troubleshooting

```
$ curl -sL "http://keyserver.ubuntu.com/pks/lookup?op=get&
↪search=0xA5D32F012649A5A9" | sudo apt-key add
```

  - lastly do another

```
$ sudo apt-update && sudo apt upgrade
```

- **Step 4**: Install datalad and everything it needs

```
$ sudo apt install datalad
```

3) Install within WSL2

The Windows Subsystem for Linux (WSL) allows Windows users to have full access to a Linux distribution within Windows. The Windows Subsystem for Linux 2 (WSL2) is the (currently pre-released) update to the WSL. If you have always used Windows be prepared for some user experience changes when using Linux compared to Windows. For one, there will be no graphical user interface (GUI). Instead, you will work inside a terminal window. This however mirrors the examples and code snippets provided in this handbook exactly. Using a proper Linux installation improves the DataLad handbook experience on Windows *greatly*. However, it comes with the downside of two filesystems that are somewhat separated. Data access to files within Linux from within Windows is problematic: Note that there will be incompatibilities between the Windows and Linux filesystems. Files that are created within the WSL for example can not be modified with Windows tools. A great resource to get started and understand the WSL is this guide[40].

Requirements:

WSL can be enabled for **64-bit** versions of **Windows 10** systems running Windows 10 Insider Preview Build 18917 or higher. You can find out how to enter the Windows Insider Program to get access to the prebuilds here[41]. To check whether your computer fulfills these requirements, open *Settings* (in the start menu) > *System* > *About*. Your version number should be at least 1903. Furthermore, your computer needs to support Hyper-V Virtualization[42].

The instructions below show you how to set up the WSL and configure it to use DataLad and its dependencies. They follow the Microsoft Documentation on the Windows Subsystem for Linux[43]. If you run into troubles during the installation, please consult the WSL troubleshooting page[44].

- **Step 1**: Enable the windows subsystem for Linux.
  - Start the Power Shell as an administrator. Run both commands below, only restart after the second one (despite being prompted after the first one already):

```
Enable-WindowsOptionalFeature -Online -FeatureName VirtualMachinePlatform
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-
↪Linux
```

- **Step 2**: Install a Debian Linux distribution

[40] https://github.com/michaeltreat/Windows-Subsystem-For-Linux-Setup-Guide/
[41] https://insider.windows.com/en-us/
[42] https://www.thomasmaurer.ch/2017/08/install-hyper-v-on-windows-10-using-powershell/
[43] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[44] https://docs.microsoft.com/en-us/windows/wsl/troubleshooting

  - To do this, visit the Microsoft store, and search for the Debian distro. We **strongly** recommend installing *Debian*, even though other distributions are available. “Get” the app, and “install” it.

- **Step 3**: Initialize the distribution
  - Launch the Subsystem either from the Microsoft store or from the Start menu. This will start a terminal. Do not worry – there is a dedicated section (*General prerequisites* (page 29)) on how to work with the terminal if you haven’t so far.
  - Upon first start, you will be prompted to enter a new UNIX username and password. Tip: chose a short name, and no spaces or special characters. The password will become necessary when you elevate a process using sudo – sudo let’s you execute a process with rights of another user, such as administrative rights, for examples when you need to install software.

- **Step 4**: Configure the WLS
  - Start the Power Shell as an administrator. To set the WSL version to WSL2, run `wsl --set-default-version 2`. Configure the distro to use WSL2 by running `wsl -l -v`. This should give an output like this:

```
    NAME          STATE                 VERSION
*   Debian         Running              2
```

- **Step 5**: Enable NeuroDebian
  - In the terminal of your distribution, run

```
$ wget -O- http://neuro.debian.net/lists/stretch.de-md.libre | sudo tee /etc/
↪apt/sources.list.d/neurodebian.sources.list
```

  - Afterwards, run

```
$ curl -sL "http://keyserver.ubuntu.com/pks/lookup?op=get&
↪search=0xA5D32F012649A5A9" | sudo apt-key add
```

  - lastly do another

```
$ sudo apt-update && sudo apt upgrade
```

- **Step 6**: Install datalad and everything it needs from NeuroDebian

```
$ sudo apt install datalad
```

Todo:
- maybe update Step 6 to use pip3 to install DataLad and git-annex.

3.2 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
```

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.

CHAPTER

FOUR

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

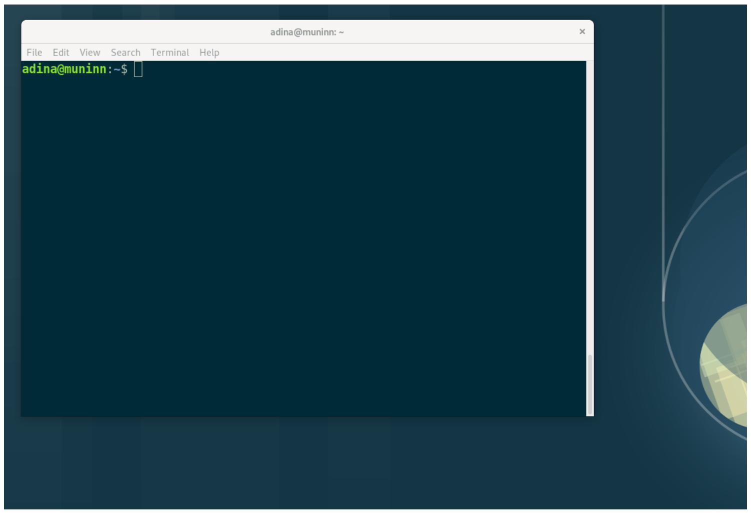
The shell (sometimes also called a terminal, console, or CLI) is an interactive, text based interface. If you have used Matlab or IPython, then you are already familiar with the basics of a command line interface.

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 Jan  9 07:54 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



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

Fig. 1: A terminal window in a standard desktop environment.

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 Jan  9 07:54 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-02
                └── bold3T
            ├── ...
            └── sub-xx
                └── bold3T
        └── structural
            └── sub-01
                └── anat
            └── sub-02
                └── anat
            ├── ...
            └── sub-xx
                └── anat
```

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 advanture!

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

CHAPTER

FIVE

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.

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

5.2 Simplified local version control workflows

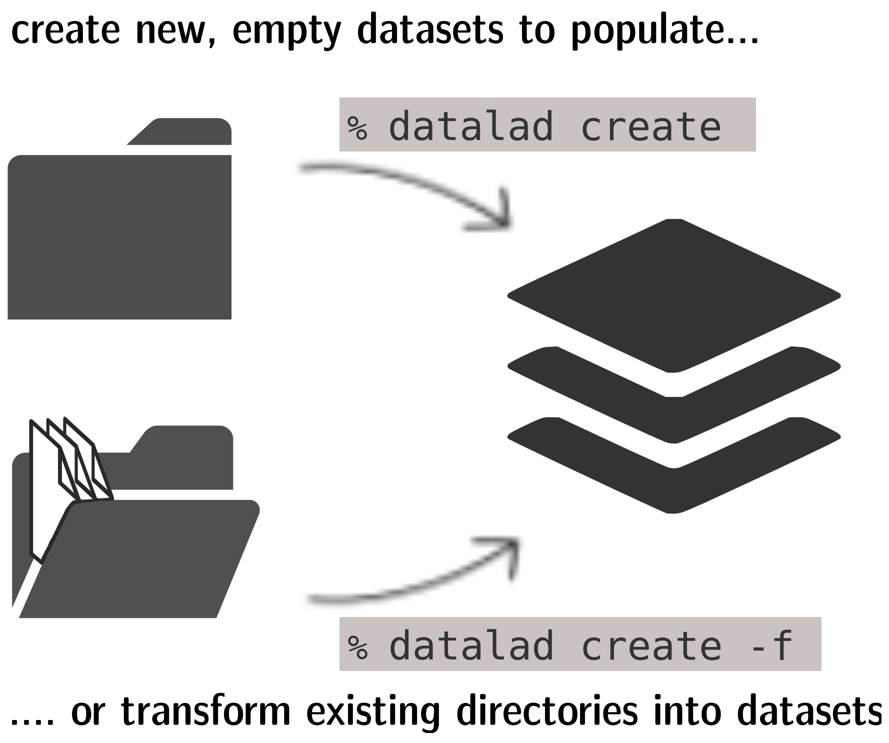
Building on top of *Git* and *git-annex*, DataLad allows you to version control arbitrarily large files in datasets.

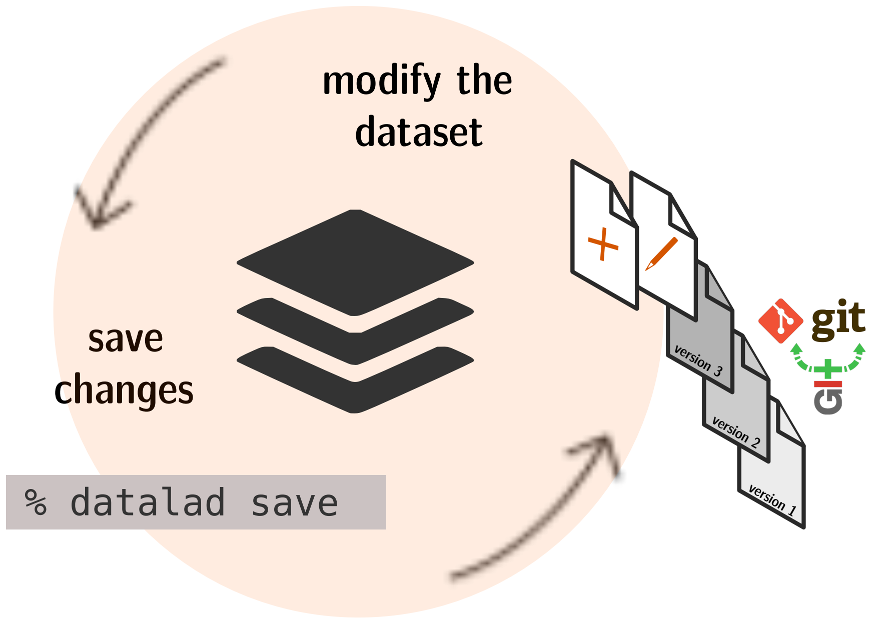
Thus, you can keep track of revisions of data of any size, and view, interact with or restore any version of your dataset's history.

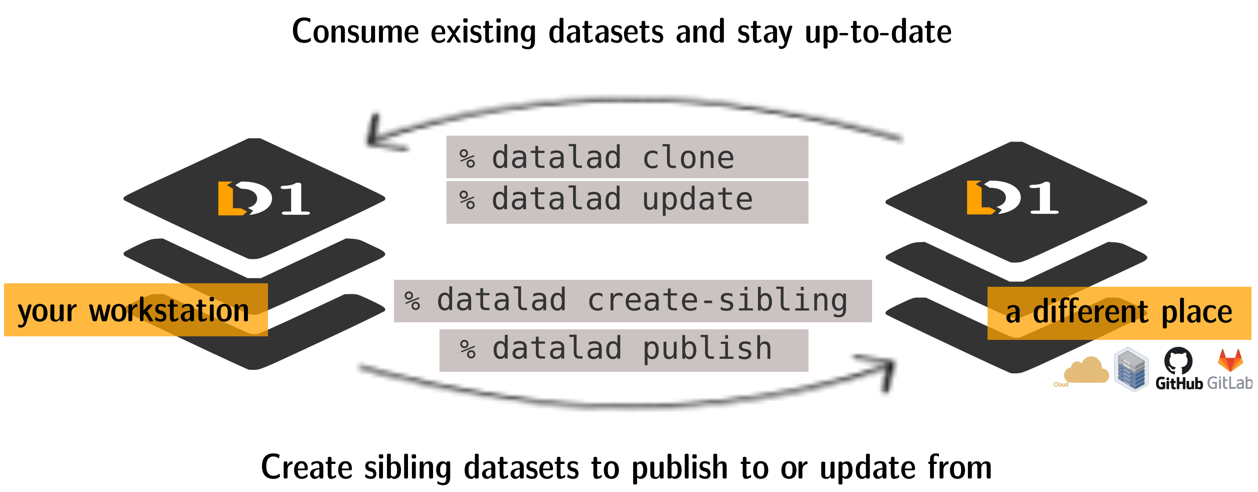
5.3 Consumption and collaboration

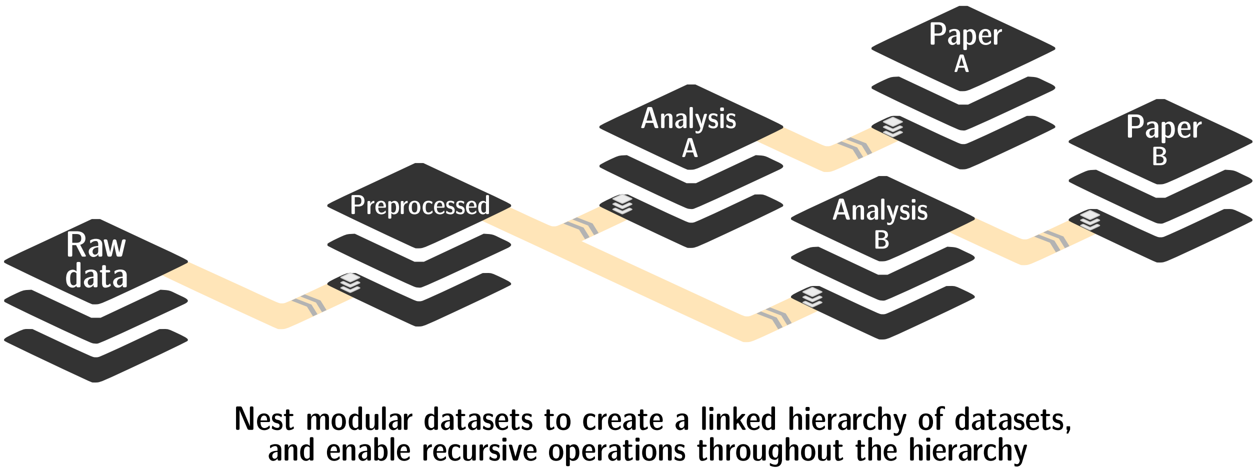
DataLad lets you consume datasets provided by others, and collaborate with them. You can install existing datasets and update them from their sources, or create sibling datasets that you can publish updates to and pull updates from for collaboration and data sharing.

Additionally, you can get access to publicly available open data collections with *the DataLad superdataset ///*.


create new, empty datasets to populate...
% datalad create
% datalad create -f
.... or transform existing directories into datasets


modify the dataset
save changes
% datalad save
version 3
version 2
version 1
git




Paper A
Analysis A
Paper B
Preprocessed
Analysis B
Raw data
Nest modular datasets to create a linked hierarchy of datasets, and enable recursive operations throughout the hierarchy

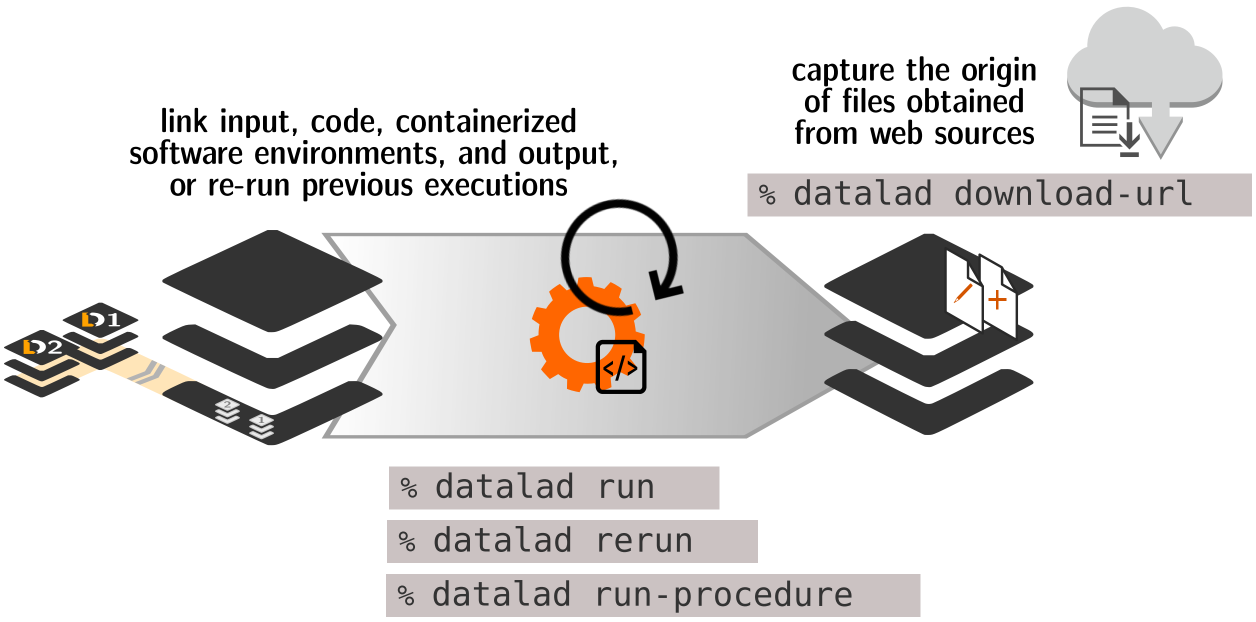
5.4 Dataset linkage

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.

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).

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?).


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

5.6 Third party service integration

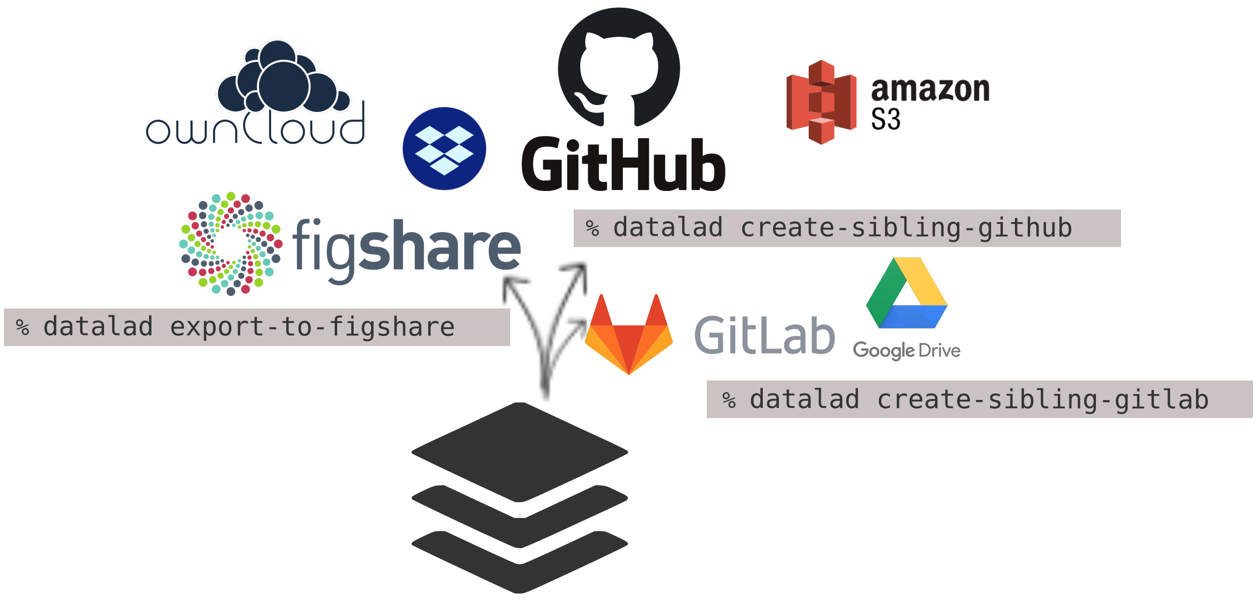
Export datasets to third party services such as GitHub[46], GitLab[47], or Figshare[48] with built-in commands.

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.

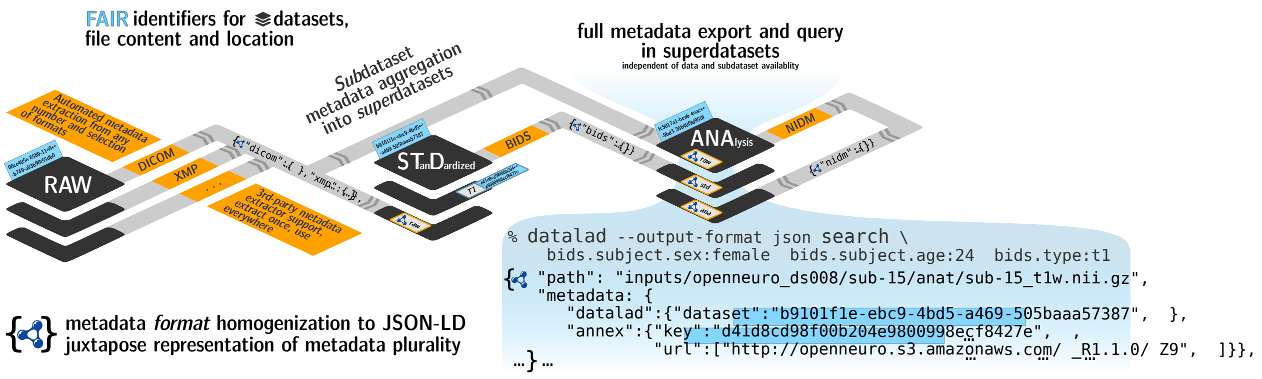
5.7 Metadata handling

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/



```
% datalad create-sibling-github
% datalad export-to-figshare
% datalad create-sibling-gitlab
```


FAIR identifiers for datasets, file content and location
Subdataset metadata aggregation into superdatasets
full metadata export and query in superdatasets
independent of data and subdataset availablity
Automated metadata extraction from any number and selection of formats
RAW
DICOM
XMP
...
3rd-party metadata extractor support, extract once, use everywhere
{"dicom":{ },"xmp":{...},
STanDardized
BIDS
T1
{"bids":{}}
ANAlysis
NIDM
{"nidm":{}}
raw
std
ana
metadata format homogenization to JSON-LD
juxtapose representation of metadata plurality

```
% datalad --output-format json search \
    bids.subject.sex:female  bids.subject.age:24  bids.type:t1
{ "path": "inputs/openneuro_ds008/sub-15/anat/sub-15_t1w.nii.gz",
  "metadata: {
     "datalad":{"dataset":"b9101f1e-ebc9-4bd5-a469-505baaa57387",  },
     "annex":{"key":"d41d8cd98f00b204e9800998ecf8427e",  ,
              "url":["http://openneuro.s3.amazonaws.com/ _R1.1.0/ Z9",  ]}},
 …} …
```

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 349), and a concise command cheat-sheet in section *DataLad cheat sheet* (page 355).

But enough of the introduction now – let's dive into the Basics!

Part III

Basics 1 – DataLad datasets

CHAPTER

SIX

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.

CHAPTER

SEVEN

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 9), 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
```

Find out more: What is the description option?

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, but later chapters (starting with *Looking without touching* (page 119)) 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] 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[54] 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 ca376f425bb13ae55305d6832694345f583753d1
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:22 2020 +0100

    Instruct annex to add text files to Git

commit d8422134ed4eb4abfc1d256ce10dcc84d7f1eb3c
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:21 2020 +0100

    [DATALAD] new dataset
```

We can see two *commit*s in the history of the repository. Each of them is identified by a unique 40 character sequence, called a *shasum*. 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). Even though these commits were produced by DataLad, in most other cases, you will have to create the commit and an informative commit message yourself.

[54] 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

Note for Git users:

`datalad create` uses **`git init`** and **`git-annex init`**. Therefore, the DataLad dataset is a Git repository. Large file content in the dataset in the annex 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.

CHAPTER

EIGHT

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[60]:

```
$ 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[55]
- An intro to Python: A byte of Python[56]

You can either visit the links and save them in `books/`, or run the following commands[61] to download the books right from the terminal:

```
$ cd books
$ wget https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.pdf/
↪download -O TLCL.pdf
$ wget https://edisciplinas.usp.br/pluginfile.php/3252353/mod_resource/content/1/b_Swaroop_
↪Byte_of_python.pdf -O byte-of-python.pdf
# get back into the root of the dataset
$ cd ../
```

(continues on next page)

[60] `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` or `brew`). For example, if you use OSX, `brew install tree` will get you this tool.

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

[56] https://edisciplinas.usp.br/pluginfile.php/3252353/mod_resource/content/1/b_Swaroop_Byte_of_python.pdf

[61] `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.

(continued from previous page)

```
2020-01-09 14:23:38 URL:https://netix.dl.sourceforge.net/project/linuxcommand/TLCL/19.01/
↪TLCL-19.01.pdf [2120211/2120211] -> "TLCL.pdf" [1]
2020-01-09 14:23:57 URL:https://edisciplinas.usp.br/pluginfile.php/3252353/mod_resource/
↪content/1/b_Swaroop_Byte_of_python.pdf [4242644/4242644] -> "byte-of-python.pdf" [1]
```

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:

```
$ 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)
```

Find out more: "Oh no! I forgot the -m option!"

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 247) 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 69e79830c5593ba363d9e38262057dea6efd634d
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:42 2020 +0100

    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..58c0629
--- /dev/null
+++ b/books/byte-of-python.pdf
```

Now this might look a bit cryptic (and honestly, tig[62] 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.

Find out more: 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.
- here you can find more guidelines: https://gist.github.com/robertpainsi/b632364184e70900af4ab688decf6f53

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

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

Note for Git users:

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 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 with that! 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[57]:

```
$ cd books
$ wget https://github.com/progit/progit2/releases/download/2.1.154/progit.pdf
$ cd ../
2020-01-09 07:51:44 URL:https://github-production-release-asset-2e65be.s3.amazonaws.com/
↪15400220/57552a00-9a49-11e9-9144-d9607ed4c2db?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-
↪Credential=AKIAIWNJYAX4CSVEH53A%2F20200109%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
↪Date=20200109T065143Z&X-Amz-Expires=300&X-Amz-
↪Signature=b9ddebee5188635d6d21f9084a03c611dc1144e277976fea5665032af73ef629&X-Amz-
↪SignedHeaders=host&actor_id=0&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 **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:
```

(continues on next page)

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

(continued from previous page)

```
add (ok: 1)
save (ok: 1)
```

Find out more: Some more on save

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. There are some ways to mitigate this: 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:

```
# lets make the output a bit more concise with the --oneline option
$ git log --oneline
e1e8af3 add reference book about git
69e7983 add books on Python and Unix to read later
ca376f4 Instruct annex to add text files to Git
d842213 [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[58]). 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, lets add a final book, a beginner's guide to bash[59], 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. Note that we line break the command with \ signs. You can copy them as they are presented here into your terminal, but in your own work you can write commands like this into a single line.

[58] https://xkcd.com/1205/
[59] http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

```
$ 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
```

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

```
$ datalad status
```

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

```
$ git log -p -n 1
commit da7d2d02b43040b213ca22ce2c958fc863f8c1f4
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:48 2020 +0100

    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
da7d2d0 add beginners guide on bash
e1e8af3 add reference book about git
69e7983 add books on Python and Unix to read later
ca376f4 Instruct annex to add text files to Git
d842213 [DATALAD] new dataset
```

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

CHAPTER

NINE

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[63]. 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`.

Find out more: 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)[64], the standard location that provides the *input* for a command. Thus, the text stream becomes the input for the cat command[65], which takes the input and writes it to "standard output" (stdout)[66].

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,

[63] https://en.wikipedia.org/wiki/Here_document
[64] https://en.wikipedia.org/wiki/Standard_streams#Standard_input_(stdin)
[65] https://en.wikipedia.org/wiki/Cat_(Unix)
[66] https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)

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.

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

```
$ 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:

```
$ 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 redirecting, 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 0023a975216764ba8affca057836236521ad0480
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:50 2020 +0100

    add note on datalad save

diff --git a/notes.txt b/notes.txt
index 3a7a1fe..bfa64d7 100644
--- a/notes.txt
+++ b/notes.txt
@@ -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 3baae1ee0e76a24307a28f0ae7086a649058abd8
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:49 2020 +0100

    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?

Find out more: 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).

CHAPTER

TEN

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[68]. 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.

[68] 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 http://longnow.org/seminars/podcast. Support the foundation by becoming a member: https://longnow.org/membership. http://longnow.org

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 a hidden section in section *Looking without touching* (page 119) 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[67]. The dataset to be installed lives on *GitHub*, at https://github.com/datalad-datasets/longnow-podcasts.git, and we can give its GitHub URL as the first positional argument. Optionally, the command also takes a second positional path as an argument – to the *destination*, i.e., 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 for 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 https://github.com/datalad-datasets/longnow-podcasts.git [1 other candidates]␣
↪into '/home/me/dl-101/DataLad-101/recordings/longnow'
[INFO]   Remote origin not usable by git-annex; setting annex-ignore
add(ok): recordings/longnow (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
install(ok): recordings/longnow (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

[67] 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.

though this is not obvious at this point in time, later chapters in this handbook will demonstrate how useful this information can be.

Find out more: 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 don't 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
```

Find out more: 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.

Note for Git users:

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:

```
$ 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
```

(continues on next page)

(continued from previous page)

```
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
```

10.1 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)
```

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 aren't even sure whether your hard drive has much space left...

But you nevertheless are curious on how to actually listen to one of these `.mp3`s 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)
```

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
```

(continues on next page)

(continued from previous page)

```
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...]
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...]
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`.

Note for Git users:

`datalad get` uses **`git annex get`** underneath the hood.

You have now experienced how easy it is to 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
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, you 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
```

(continues on next page)

(continued from previous page)

```
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)
```

Note: 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[69]) 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 111).

[69] You can also upgrade your file manager to display file types in a DataLad datasets (e.g., with the git-annex-turtle extension[70] for Finder)

CHAPTER

ELEVEN

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 2
commit 2fcef51461b5c8e0c1fcf1ed6511035fb3c79509
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:51:52 2020 +0100

    [DATALAD] Recorded changes

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

(continues on next page)

(continued from previous page)

```
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
```

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
43fdea1 Add script to generate a README from DataLad metadata
997e07a Update aggregated metadata
8031017 Consolidate all metadata-related files under .datalad
8053eed Add annexed feed logos
1a396a6 Prepare to annex big feed logos
75d7f3f Rename metadata directory
5dd7772 Manually place extracted metadata in Git
b9c517e Make sure extracted metadata is directly in Git
0553111 content removed from git annex
39226e9 Update aggregated metadata
740fa14 [DATALAD RUNCMD] Update from feed
61f46fc Add base dataset metadata
3e96466 More diff-able
979bd25 Single update maintainer script
ead809e Be resilient with different delimiters
9bece59 Add duration to the metadata
f0831b9 Script to convert the RSS feed metadata into JSON-LD metadata
```

(continues on next page)

(continued from previous page)

```
e64d00f Prepare for addition of RSS feed metadata on episodes
e1bf31e [DATALAD RUNCMD] Update SALT series
21d9290 [DATALAD RUNCMD] Update Interval seminar series
7f36dea Update from feed
ff00713 Update from feed
a052af9 Include publication date in the filename
9f3127f Import Interval feed
b81bdea Import SALT feed
3d0dc8f [DATALAD] new dataset
8df130b [DATALAD] Set default backend for all files to be MD5E
```

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.

Find out more: 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:

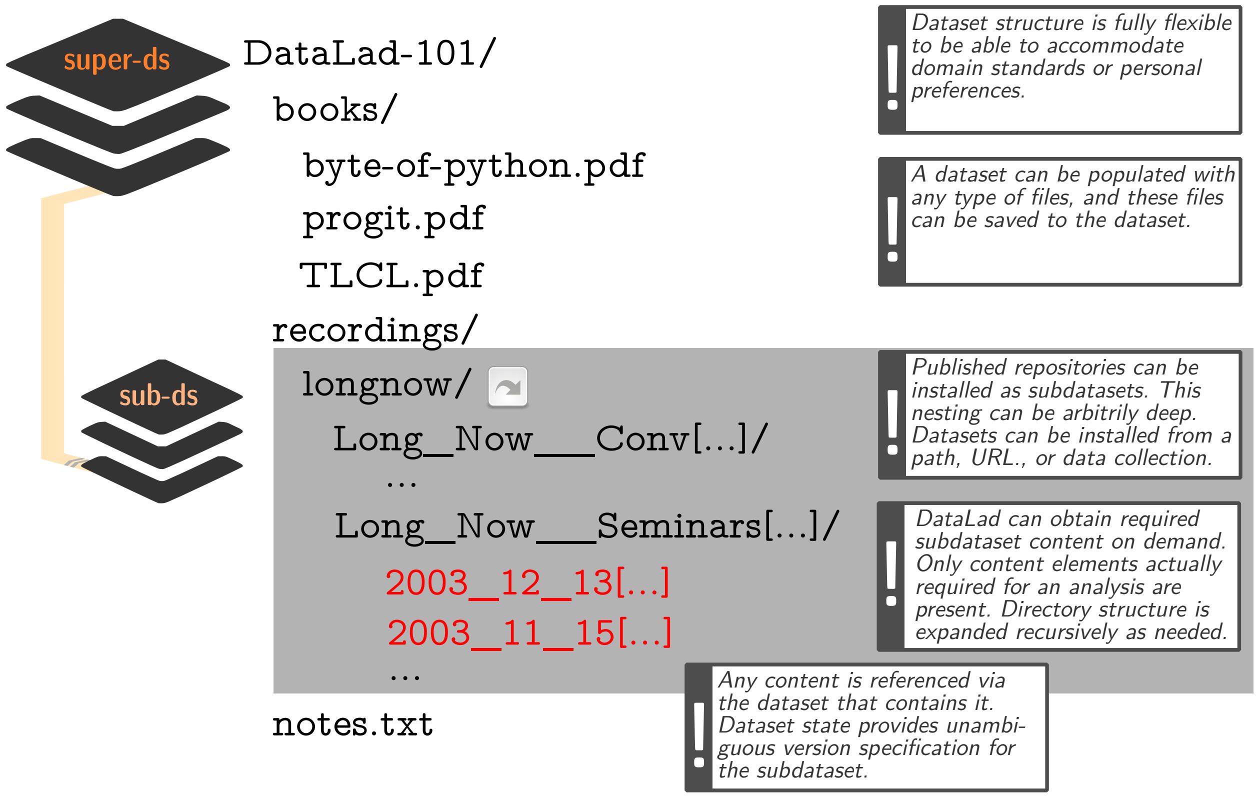
```
$ git -C recordings/longnow log --oneline
```

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 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 ../../
```


super-ds
DataLad-101/
books/
byte-of-python.pdf
progit.pdf
TLCL.pdf
recordings/
sub-ds
longnow/
Long_Now__Conv[...]/
...
Long_Now__Seminars[...]/
2003_12_13[...]
2003_11_15[...]
...
notes.txt
Dataset structure is fully flexible to be able to accommodate domain standards or personal preferences.
A dataset can be populated with any type of files, and these files can be saved to the dataset.
Published repositories can be installed as subdatasets. This nesting can be arbitrily deep. Datasets can be installed from a path, URL., or data collection.
DataLad can obtain required subdataset content on demand. Only content elements actually required for an analysis are present. Directory structure is expanded recursively as needed.
Any content is referenced via the dataset that contains it. Dataset state provides unambiguous version specification for the subdataset.

CHAPTER

TWELVE

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*:
- **`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

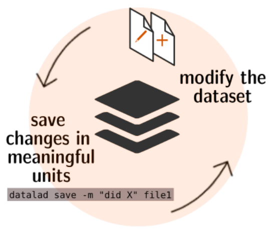

modify the dataset
save changes in meaningful units
datalad save -m "did X" file1

Fig. 1: A simple, local version control workflow with DataLad.

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 61)), 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.

12.1 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 message*s 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.

Part IV

Basics 2 – Datalad, Run!

CHAPTER

THIRTEEN

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[71]. 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 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.

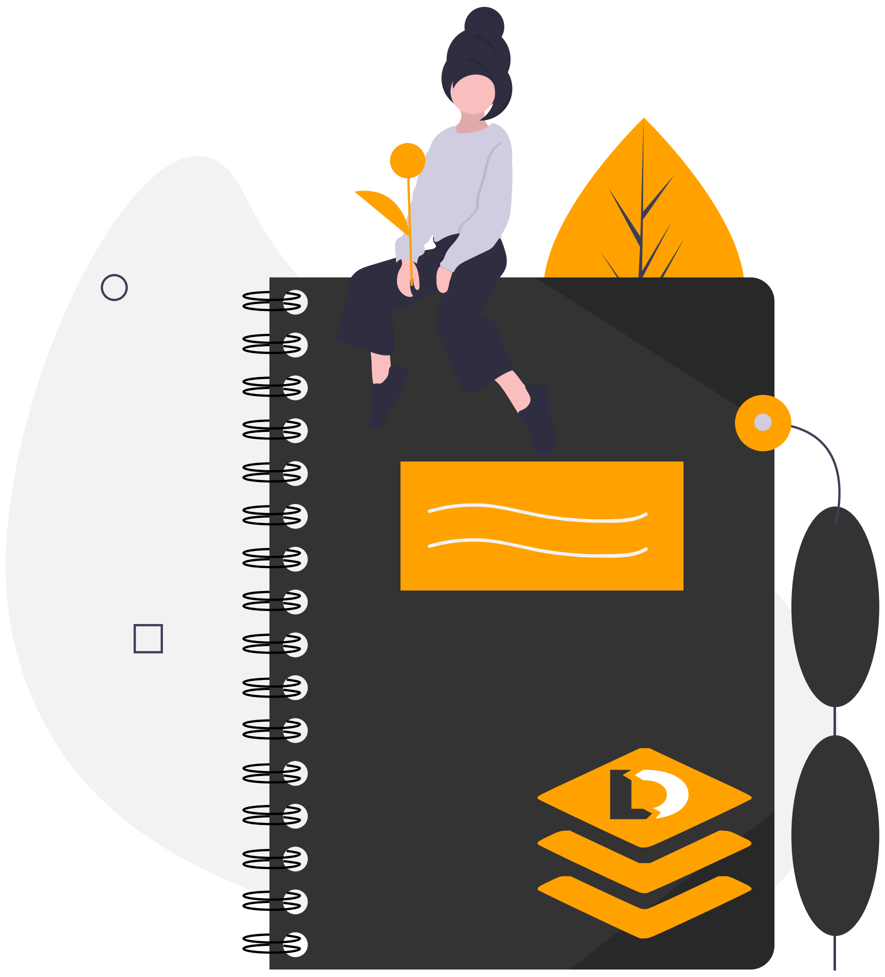
To do this, we're following a best practice that will reappear in the later section on YODA principles: Collecting all additional scripts that work with content of a subdataset *outside* of this subdataset, in a dedicated `code/` directory, and collating the output of the execution of these scripts *outside* 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
```

(continues on next page)

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



(continued from previous page)

```
.
├── 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.

```
$ 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)
```

Find out more: 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" summary. 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
commit 87fde6aa2aa9e696730d4ffcf4313a2b5cb9a790
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:52:00 2020 +0100

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "71f03bec-32ac-11ea-b7a4-e86a64c8054c",
     "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
```

(continues on next page)

(continued from previous page)

```
index 0000000..f691b53
--- /dev/null
+++ 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
87fde6a [DATALAD RUNCMD] create a list of podcast titles
5b391e9 Add short script to write a list of podcast speakers and titles
445c53e Add note on datalad clone
2fcef51 [DATALAD] Recorded changes
```

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.

CHAPTER

FOURTEEN

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
2005-01-15        James Carse  Religious War In Light of the Infinite Game
2005-02-26        Roger Kennedy  The Political History of North America from 25 000 BC to 12␣
↪000 AD
2005-03-12        Spencer Beebe  Very Long term Very Large scale Biomimicry
2005-04-09        Stewart Brand  Cities   Time
2005-06-11        Robert Neuwirth  The 21st Century Medieval City
2005-07-16        Jared Diamond  How Societies Fail And Sometimes Succeed
2005-08-13        Robert Fuller  Patient Revolution  Human Rights Past and Future
2005-09-24        Ray Kurzweil  Kurzweil s Law
2005-10-06        Esther Dyson  Freeman Dyson  George Dyson  The Difficulty of Looking Far␣
↪Ahead
2005-11-15        Clay Shirky  Making Digital Durable  What Time Does to Categories
2005-12-10        Sam Harris  The View from the End of the World
2006-01-14        Ralph Cavanagh  Peter Schwartz  Nuclear Power  Climate Change and the Next␣
↪10 000 Years
2006-02-14        Stephen Lansing  Perfect Order  A Thousand Years in Bali
2006-03-11        Kevin Kelly  The Next 100 Years of Science  Long term Trends in the␣
↪Scientific Method.
2006-04-15        Jimmy Wales  Vision  Wikipedia and the Future of Free Culture
```

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:

```
$ cat << EOT >| code/list_titles.sh
for i in recordings/longnow/Long_Now*/*.mp3; do
   # 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 0a15563cb64708782078f62afa168bad70285704
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 18:07:32 2020 +0100

    BF: list both directories content

commit f8cfd6644a2a1a1a5bc445558b30b3b1234471c6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 18:07:30 2020 +0100

    [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 f8cfd6644a2a1a1a5bc445558b30b3b1234471c6
[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 d6b68a1b7fa23c684d635d2dacedf75126759c59
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 18:07:34 2020 +0100

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [
      "f8cfd6644a2a1a1a5bc445558b30b3b1234471c6"
     ],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "55c10900-3302-11ea-b7a4-e86a64c8054c",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [],
```

(continues on next page)

(continued from previous page)

```
  "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[72]):

```
$ 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
+2018-02-01       Humanity and the Deep Ocean  James Nestor
+2018-03-01       Our Future in Algorithm Farming  Mike Kuniavsky
+2018-04-18       The Organized Pursuit of Knowledge  Margaret Levi
+2018-08-15       Facts  Feelings and Stories  How to Motivate Action on Climate Change ␣
↪Shahzeen Attari
```

(continues on next page)

[72] The section *Back and forth in time* (page 247) will elaborate more on common *Git* commands and terminology.

(continued from previous page)

```
+2019-03-26       Charting the High Frontier of Space  Ed Lu
+2019-04-04       The Science of Climate Fiction  Can Stories Lead to Social Action   James␣
↪Holland Jones
+2019-04-10       The Spirit Singularity  Science and the Afterlife at the Turn of the 20th␣
↪Century  Hannu Rajaniemi
+2019-04-18       The Evolving Science of Behavior Change  Christopher Bryan
+2019-04-30       Siberia  A Journey to the Mammoth Steppe  Stewart Brand  Kevin Kelly ␣
↪Alexander Rose
+2019-05-06       Can Nationalism be a Resource for Democracy   Maya Tudor
+2019-05-14       Growing Up Ape  The Long term Science of Studying Our Closest Living␣
↪Relatives  Elizabeth  Lonsdorf
+2019-05-21       Time Poverty Amidst Digital Abundance  Judy Wajcman
+2019-06-07       A Foundation of Trust  Building a Blockchain Future  Brian Behlendorf
+2019-07-12       Learning From Le Guin  Kim Stanley Robinson
 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
```

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`:

```
$ 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 `--`):

```
$ git log -- recordings/podcasts.tsv
commit d6b68a1b7fa23c684d635d2dacedf75126759c59
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 18:07:34 2020 +0100

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [
      "f8cfd6644a2a1a1a5bc445558b30b3b1234471c6"
     ],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "55c10900-3302-11ea-b7a4-e86a64c8054c",
     "exit": 0,
     "extra_inputs": [],
     "inputs": [],
     "outputs": [],
     "pwd": "."
    }
    ^^^ Do not change lines above ^^^

commit f8cfd6644a2a1a1a5bc445558b30b3b1234471c6
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 18:07:30 2020 +0100

    [DATALAD RUNCMD] create a list of podcast titles

    === Do not change lines below ===
    {
     "chain": [],
     "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv",
     "dsid": "55c10900-3302-11ea-b7a4-e86a64c8054c",
     "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)
```

CHAPTER

FIFTEEN

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[73]. 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`.

[73] https://imagemagick.org/index.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/2874.
convert-im6.q16: no images defined `recordings/salt_logo_small.jpg' @ error/convert.c/
↪ConvertImageCommand/3258.
[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: command 'convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/
↪logo_salt.jpg recordings/salt_logo_small.jpg' failed with exitcode 1
Failed to run 'convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.
↪jpg recordings/salt_logo_small.jpg' under '/home/me/dl-101/DataLad-101'. Exit code=1.
```

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[75]. 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.

[75] 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)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
get(ok): recordings/longnow/.datalad/feed_metadata/logo_salt.jpg (file) [from web...]
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 61) 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.

Find out more: What if there are several inputs?

Often, a command needs several inputs. In principle, every input 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.

15.1 If outputs already exist…

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" \
```

(continues on next page)

(continued from previous page)

```
"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/2874.
[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: command 'convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/
↪logo_salt.jpg recordings/salt_logo_small.jpg' failed with exitcode 1
Failed to run 'convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.
↪jpg recordings/salt_logo_small.jpg' under '/home/me/dl-101/DataLad-101'. Exit code=1.
```

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 for help. 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:

```
$ 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 (symlink)
```

"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.

Find out more: 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[74], a neuro-imaging tool. The easiest way to specify this type of output is the directory name and a *globbing* character, such as -o directory/*. And, just as for -i/--input, you could use multiple --output specifications.

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:

```
$ datalad run -m "Resize logo for slides" \
--input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \
--output "recordings/interval_logo_small.jpg" \
```

(continues on next page)

[74] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki

(continued from previous page)

```
"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)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
get(ok): recordings/longnow/.datalad/feed_metadata/logo_interval.jpg (file) [from web...]
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
```

15.2 Placeholders

Just after writing this note, you have 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.

Find out more: 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.

Find out more: ... wait, what if I need a { or } character 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 `}}`.

CHAPTER

SIXTEEN

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]
```

16.1 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 on *Populate a dataset* (page 49): 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)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
unlock(ok): recordings/interval_logo_small.jpg (file)
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 DataLad 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)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
unlock(ok): recordings/salt_logo_small.jpg (file)
add(ok): recordings/salt_logo_small.jpg (file)
```

(continues on next page)

(continued from previous page)

```
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 doesn't 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).

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
commit cd4e9c17d3a219c33fb9e3ba3ce35de831164a06
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:52:14 2020 +0100

    [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": "71f03bec-32ac-11ea-b7a4-e86a64c8054c",
     "exit": 0,
     "extra_inputs": [],
```

(continues on next page)

(continued from previous page)

```
     "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
```

CHAPTER

SEVENTEEN

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.
- 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.

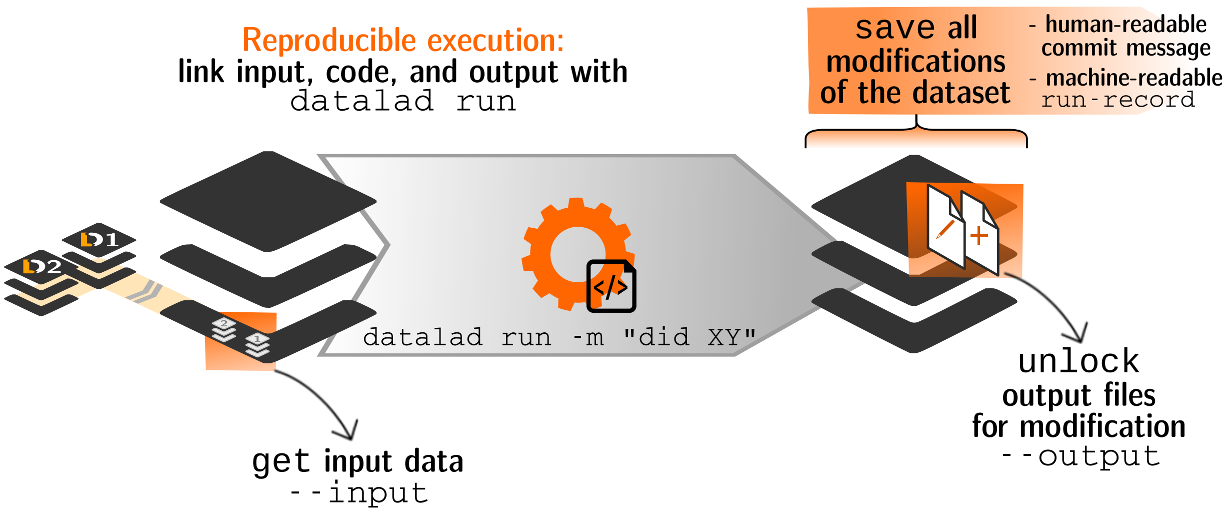

Reproducible execution:
link input, code, and output with
datalad run
save all modifications of the dataset
- human-readable commit message
- machine-readable run-record
datalad run -m "did XY"
unlock output files for modification
--output
get input data
--input

Fig. 1: Overview of `datalad run`.

17.1 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.

Part V

Basics 3 – Under the hood: git-annex

CHAPTER

EIGHTEEN

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 45). 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:

```
$ git log --reverse --oneline
d842213 [DATALAD] new dataset
```

(continues on next page)

(continued from previous page)

```
ca376f4 Instruct annex to add text files to Git
69e7983 add books on Python and Unix to read later
e1e8af3 add reference book about git
da7d2d0 add beginners guide on bash
3baae1e Add notes on datalad create
0023a97 add note on datalad save
2fcef51 [DATALAD] Recorded changes
445c53e Add note on datalad clone
5b391e9 Add short script to write a list of podcast speakers and titles
87fde6a [DATALAD RUNCMD] create a list of podcast titles
f2c608e BF: list both directories content
3d0c225 [DATALAD RUNCMD] create a list of podcast titles
cec87ae add note datalad and git diff
a2055bf add note on basic datalad run and datalad rerun
4291a9f [DATALAD RUNCMD] convert -resize 400x400 recordings/longn...
b71548b resized picture by hand
97f36f1 [DATALAD RUNCMD] convert -resize 450x450 recordings/longn...
3f06057 add additional notes on run options
cfd6f24 [DATALAD RUNCMD] Resize logo for slides
cd4e9c1 [DATALAD RUNCMD] Resize logo for slides
b64a92b add note on clean datasets
```

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.

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[76]."

"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.

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 100MB 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`

[76] If you can't wait to read about `.gitattributes` and other configuration files, jump ahead to chapter "Tuning datasets to your needs", starting with section *DIY configurations* (page 151).

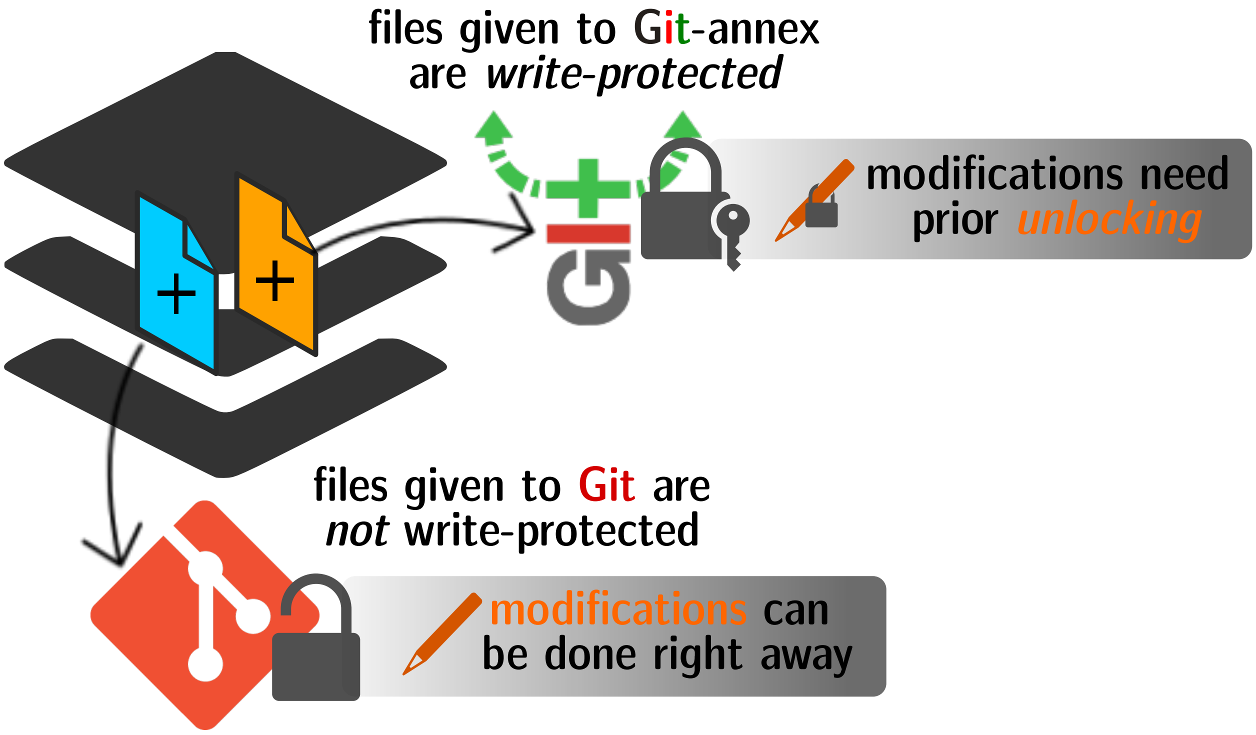


Fig. 1: A simplified overview of the tools that manage data in your dataset.

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 branch*es. 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. BUT: it is possible, and in many cases useful, and 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.

CHAPTER

NINETEEN

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_files.py`, 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 textfile 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`, requires – spoiler: at least in our current type of dataset – 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:

```
# 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/F1/Wz/MD5E-s4242644--
↪f4e1c8ebfb5c89a69ff6d268eb2e63e3.pdf/MD5E-s4242644--f4e1c8ebfb5c89a69ff6d268eb2e63e3.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 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 45) contained a note that strongly advised that you to not temper 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[81].

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, 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.

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>` (Note: exchange `evince` with your standard PDF reader).

```
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?

[81] 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.

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 Jan  9 07:51 .
drwxr-xr-x 7 adina adina 4.0K Jan  9 07:51 ..
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 Apr 19  2017 byte-of-python.pdf -> ../.git/annex/objects/F1/Wz/
↪MD5E-s4242644--f4e1c8ebfb5c89a69ff6d268eb2e63e3.pdf/MD5E-s4242644--
↪f4e1c8ebfb5c89a69ff6d268eb2e63e3.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.

Note for Git users:

Small symlinks can be written very very fast when switching branches, as opposed to copying and deleting huge data files.

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. If you are interested in why these paths look so weird, and what all of this has to do with data integrity, you can check out the hidden section below.

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 229)).

Find out more: more about paths, checksums, object trees, and data integrity

But why does the target path to the object tree needs to be so cryptic? Does someone want to create maximal confusion with this naming? Can't it be . . . more *readable*?

Its not malicious intent that leads to these paths and file names. Its checksums. And they are quite readable – just not for humans, but git-annex. Understanding the next section is completely irrelevant for the subsequent sections of the book. But it can help to establish trust in that your data are safely stored and tracked, and it can get certainly helpful should you be one of those people that always want to understand things in depth. Also, certain file management operations can be messy – for example, when you attempt to move a subdirectory (more on this in a dedicated section *Miscellaneous file system operations* (page 229)) it can break symlinks, and you need to take appropriate actions to get the dataset back into a clean state. Understanding more about the object tree can help to understand such problems, and knowing bits of the git-annex basics can make you more confident in working with your datasets.

So how do these paths and names come into existence?

When a file is annexed, git-annex generates a *key* 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 two files have identical contents, and whether file content changed.

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. In principle, therefore, the hash function simply transforms a content of any size into a string with fixed length.

The important aspect of a hash function is that it will generate the same hash for the same file content, but once file content changes, the generated hash will also look different. If two files are turned into identical character strings, the content in these files is thus identical. Therefore, if two files have the same symlink, and thus link the same file in the object-tree, they are identical in content. If you have many copies of the same data in your dataset, the object tree will contain only one instance of that content, and all copies will symlink to it, thus saving disk space. But furthermore, the file name also becomes a way of ensuring data integrity. File content can not be changed without git-annex noticing, because the symlink to the file content will change. If you want to read more about the computer science basics about about hashes check out the Wikipedia page here[77].

This key (or *checksum*) is the last part of the name of the file the symlink links to (in which the actual data content is stored). The extension (e.g., `.pdf`) is appended because some operating systems (Windows) need this information. The key is also one of the subdirectory names in the path. This subdirectory adds an important feature to the *object-tree*: It revokes the users *permissions* to modify it. This two-level structure is implemented because it helps to prevent accidental deletions and changes, and this information will be helpful to understand some file system management

[77] https://en.wikipedia.org/wiki/Hash_function

operations (see section *Miscellaneous file system operations* (page 229)), for example deleting a subdataset.

```
# 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
```

```
# compare it to the checksum (here of type md5sum) of the PDF file and the subdirectory name
$ md5sum TLCL.pdf
06d1efcb05bb2c55cd039dab3fb28455  TLCL.pdf
```

There are different hash functions available. Depending on which is used, the resulting *checksum* has a certain length and structure. By default, DataLad uses `MD5E` checksums, but should you want to, you can change this default to one of many other types[78]. The first part of the file name actually states which hash function is used. The reason why MD5E is used is because it is comparatively short – thus it is possible to share your datasets also with users on operating systems that have restrictions on total path lengths (Windows). Therefore, refrain from changing this default if you are on Windows, or want Windows user to be able to use your dataset.

By now we know where almost all parts of the file name derived from – the 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`

As a last puzzle piece to shed some light onto the path in the object tree, there are two more directories on top of the subdirectory named after the checksum, just after `.git/annex/objects/`, consisting of two letters each. These two letters are also 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).

In summary, you now know a great deal about git-annex and the object tree. Maybe you are as amazed as we are about some of the ingenuity used behind the scenes. In any case, this section was hopefully insightful, and not confusing. If you are still curious about git-annex, you can check out its documentation[79].

19.1 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 229) 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 61) we explored the file hierarchy in an installed subdataset that contained many annexed `mp3` files. Upon the initial **`datalad clone`**, the annexed

[78] https://git-annex.branchable.com/backends/
[79] https://git-annex.branchable.com/git-annex/

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 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 seperate ways: Upgrade your file manager to display file types in a DataLad datasets (e.g., the git-annex-turtle extension[80] for Finder). Alternatively, use the **ls** command in a terminal instead of a file manager GUI.

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

```
$ cd ../
```

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

Part VI

Basics 4 – Collaboration

CHAPTER

TWENTY

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 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.

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 61)). 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 ../DataLad-101 into '/home/me/dl-101/mock_user/DataLad-101'
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
```

(continues on next page)

(continued from previous page)

```
│   ├── byte-of-python.pdf -> ../.git/annex/objects/F1/Wz/MD5E-s4242644--
↪f4e1c8ebfb5c89a69ff6d268eb2e63e3.pdf/MD5E-s4242644--f4e1c8ebfb5c89a69ff6d268eb2e63e3.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/36/jF/MD5E-s100877--
↪0fea9537f9fe255d827e4401a7d539e7.jpg/MD5E-s100877--0fea9537f9fe255d827e4401a7d539e7.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)
  	0b6bef5c-68c4-465f-8d32-c00ffa64dcfb -- 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. Luckily, there is a more human-readable piece of text next to it. 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!"

Find out more: What is this location, and what if I provided a description?

Back in the very first section of the Basics, *Create a dataset* (page 45), 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't 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
        0b6bef5c-68c4-465f-8d32-c00ffa64dcfb -- 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 /home/me/dl-101/DataLad-101/recordings/longnow [2 other candidates] into '/
↪home/me/dl-101/mock_user/DataLad-101/recordings/longnow'
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/F1/Wz/MD5E-s4242644--
↪f4e1c8ebfb5c89a69ff6d268eb2e63e3.pdf/MD5E-s4242644--f4e1c8ebfb5c89a69ff6d268eb2e63e3.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
```

(continues on next page)

(continued from previous page)

```
├── code
│   └── list_titles.sh
├── notes.txt
└── recordings
    ├── interval_logo_small.jpg -> ../.git/annex/objects/36/jF/MD5E-s100877--
↪0fea9537f9fe255d827e4401a7d539e7.jpg/MD5E-s100877--0fea9537f9fe255d827e4401a7d539e7.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
    │   │   ├── 2017_07_03__Transforming_Perception__One_Sense_at_a_Time__Kara_Platoni.mp3 ->␣
↪../.git/annex/objects/J6/88/MD5E-s62941770--77ae65e0f84c4b1fbefe74183284c305.mp3/MD5E-
↪s62941770--77ae65e0f84c4b1fbefe74183284c305.mp3
    │   │   ├── 2017_08_01__How_Climate_Will_Evolve_Government_and_Society__Kim_Stanley_
↪Robinson.mp3 -> ../.git/annex/objects/kw/PF/MD5E-s60929439--
↪86a30b6bab51e59af52ca8aa6684498f.mp3/MD5E-s60929439--86a30b6bab51e59af52ca8aa6684498f.mp3
    │   │   ├── 2017_09_01__Envisioning_Deep_Time__Jonathon_Keats.mp3 -> ../.git/annex/
↪objects/W4/2q/MD5E-s57113552--82a985abe7fa362e29e4ffa3a9951cc3.mp3/MD5E-s57113552--
↪82a985abe7fa362e29e4ffa3a9951cc3.mp3
    │   │   ├── 2017_10_01__Thinking_Long_term_About_the_Evolving_Global_Challenge__The_
↪Refugee_Reality.mp3 -> ../.git/annex/objects/81/qF/MD5E-s78362767--
↪5b077807c50d1fa02bebd399ec1431e0.mp3/MD5E-s78362767--5b077807c50d1fa02bebd399ec1431e0.mp3
    │   │   ├── 2017_11_01__The_Web_In_An_Eye_Blink__Jason_Scott.mp3 -> ../.git/annex/objects/
↪03/4v/MD5E-s64398689--049e8d1c9288d201b275331afb71b316.mp3/MD5E-s64398689--
↪049e8d1c9288d201b275331afb71b316.mp3
    │   │   ├── 2017_12_01__Ideology_in_our_Genes__The_Biological_Basis_for_Political_Traits__
↪Rose_McDermott.mp3 -> ../.git/annex/objects/x0/2j/MD5E-s59979926--
↪05127d163371d1152b72d98263d7848a.mp3/MD5E-s59979926--05127d163371d1152b72d98263d7848a.mp3
    │   │   ├── 2017_12_07__Can_Democracy_Survive_the_Internet___Nathaniel_Persily.mp3 -> ../.
↪git/annex/objects/5M/Pv/MD5E-s64541470--64960bf95544bc76ed564b541ebb36bc.mp3/MD5E-
↪s64541470--64960bf95544bc76ed564b541ebb36bc.mp3
    │   │   ├── 2018_01_02__The_New_Deal_You_Don_t_Know__Louis_Hyman.mp3 -> ../.git/annex/
↪objects/MZ/MP/MD5E-s61802477--8c3056079a4d3bfe1adbbf0195d57f3c.mp3/MD5E-s61802477--
↪8c3056079a4d3bfe1adbbf0195d57f3c.mp3
    │   │   ├── 2018_02_01__Humanity_and_the_Deep_Ocean__James_Nestor.mp3 -> ../.git/annex/
↪objects/3G/5v/MD5E-s55707819--6bb054946ca3e3e95fd1b1792693706c.mp3/MD5E-s55707819--
↪6bb054946ca3e3e95fd1b1792693706c.mp3
```

(continues on next page)

(continued from previous page)

```
    │   │   ├── 2018_03_01__Our_Future_in_Algorithm_Farming__Mike_Kuniavsky.mp3 -> ../.git/
↪annex/objects/GJ/J2/MD5E-s70246964--5a9f4538aa4d7bc3163067a9e7f093ca.mp3/MD5E-s70246964--
↪5a9f4538aa4d7bc3163067a9e7f093ca.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 69) 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[82]: The `--recursion-limit` option let's you specify how many levels

[82] 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. The configuration is called `datalad-recursiveinstall = skip` and it is made on a subdataset specific basis to the `.gitmodules` file of the superdataset. The chapter "Tuning datasets to your needs", starting in section *DIY configurations* (page 151), will talk about the details of configurations and the `.gitmodules` file. Below, however, is a minimally functional example

of subdatasets should be installed together with the first subdataset:

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

Find out more: datalad clone versus datalad install

You may remember from section *Install datasets* (page 61) 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 61) 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!

```
# 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
```

(continues on next page)

on how to apply the configuration and how it works:

```
# 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)


# 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)
```

(continued from previous page)

```
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)
```

Note for Git users:

A dataset that is installed from an existing source, e.g., a path or URL, it the DataLad equivalent of a *clone* in Git.

CHAPTER

TWENTYONE

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 subdatasets 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
        b554bca4-7ff5-4d5e-86a8-fd8a7726fbb2 -- me@muninn:~/dl-101/DataLad-101/recordings/
↪longnow [origin]
           da3bf937-5bd2-43ea-a07b-bcbe71f3b875 -- mih@medusa:/tmp/seminars-on-longterm-
↪thinking

  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[83].

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.

[83] 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.

```
# 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)
```

CHAPTER

TWENTYTWO

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.

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
b64a92b add note on clean datasets
cd4e9c1 [DATALAD RUNCMD] Resize logo for slides
cfd6f24 [DATALAD RUNCMD] Resize logo for slides
3f06057 add additional notes on run options
97f36f1 [DATALAD RUNCMD] convert -resize 450x450 recordings/longn...
b71548b resized picture by hand
4291a9f [DATALAD RUNCMD] convert -resize 400x400 recordings/longn...
a2055bf add note on basic datalad run and datalad rerun
cec87ae add note datalad and git diff
3d0c225 [DATALAD RUNCMD] create a list of podcast titles
```

(continues on next page)

(continued from previous page)

```
f2c608e BF: list both directories content
87fde6a [DATALAD RUNCMD] create a list of podcast titles
5b391e9 Add short script to write a list of podcast speakers and titles
445c53e Add note on datalad clone
2fcef51 [DATALAD] Recorded changes
0023a97 add note on datalad save
3baae1e Add notes on datalad create
da7d2d0 add beginners guide on bash
e1e8af3 add reference book about git
69e7983 add books on Python and Unix to read later
ca376f4 Instruct annex to add text files to Git
d842213 [DATALAD] new dataset
```

Ah, there it is, the second most recent commit. Just as already done in section *DataLad, Re-Run!* (page 83), take this shasum and plug it into a **datalad rerun** command:

```
$ datalad rerun cd4e9c17d3a219c33fb9e3ba3ce35de831164a06
[INFO] Making sure inputs are available (this may take some time)
[WARNING] no content present; cannot unlock [unlock(/home/me/dl-101/mock_user/DataLad-101/
↪recordings/salt_logo_small.jpg)]
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
get(ok): recordings/longnow/.datalad/feed_metadata/logo_salt.jpg (file) [from origin...]
remove(ok): recordings/salt_logo_small.jpg
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.

CHAPTER

TWENTYTHREE

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`:

```
# we are inside the installed copy
$ cat notes.txt
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.

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.

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.

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
```

(continues on next page)

(continued from previous page)

```
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!

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.

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 path=/home/me/dl-101/mock_user/DataLad-101>
[INFO] Applying updates to <Dataset path=/home/me/dl-101/mock_user/DataLad-101>
update(ok): . (dataset)
```

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:

```
$ cat notes.txt
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.

The command 'datalad clone URL/PATH [PATH]'
```

(continues on next page)

(continued from previous page)

```
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.

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.

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!

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.

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.

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>"

The command "git annex whereis PATH" lists the repositories that have
the file content of an annexed file. When using "datalad get" to retrieve
```

(continues on next page)

(continued from previous page)

```
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.

Note for Git users:

`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.

CHAPTER

TWENTYFOUR

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[84] he might have found very useful additional information, such as the ascii-cast on dataset nesting[85]. 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[86] 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 45) 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
```

(continues on next page)

[84] https://www.datalad.org/
[85] https://www.datalad.org/for/git-users
[86] https://raw.githubusercontent.com/datalad/datalad.org/7e8e39b1f08d0a54ab521586f27ee918b4441d69/content/asciicast/seamless_nested_repos.sh

(continued from previous page)

```
[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)
```

Run a quick datalad status:

```
$ datalad status
```

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 d4cce9c59acef0bd9d0a26fa6a11261a272015ca
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:52:24 2020 +0100

    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 @@
+#!/bin/bash
+# This script was converted using cast2script from:
+# docs/casts/seamless_nested_repos.sh
+set -e -u
+export GIT_PAGER=cat
+
+# DataLad provides seamless management of nested Git repositories...
+
+# Let's create a dataset
+datalad create demo
+cd demo
+
+# A DataLad dataset is just a Git repo with some initial configuration
+git log --oneline
+
+# We can generate nested datasets, by telling DataLad to register a
+# new dataset in a parent dataset
```

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.

Note for Git users:

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.

Find out more: 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 135). 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 path=/home/me/dl-101/DataLad-101>
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:

```
$ 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:

```
$ 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
-
-# DataLad provides seamless management of nested Git repositories...
-
-# Let's create a dataset
-datalad create demo
-cd demo
-
-# A DataLad dataset is just a Git repo with some initial configuration
-git log --oneline
-
-# We can generate nested datasets, by telling DataLad to register a
-# new dataset in a parent dataset
-datalad create -d . sub1
-
-# A subdataset is nothing more than regular Git submodule
-git submodule
-
-# Of course subdatasets can be nested
-datalad create -d . sub1/justadir/sub2
-
-# Unlike Git, DataLad automatically takes care of committing all
-# changes associated with the added subdataset up to the given
-# parent dataset
-git status
-
-# Let's create some content in the deepest subdataset
-mkdir sub1/justadir/sub2/anotherdir
-touch sub1/justadir/sub2/anotherdir/afile
-
-# Git can only tell us that something underneath the top-most
-# subdataset was modified
-git status
-
-# DataLad saves us from further investigation
-datalad diff -r
-
-# Like Git, it can report individual untracked files, but also across
-# repository boundaries
-datalad diff -r --report-untracked all
-
```

(continues on next page)

(continued from previous page)

```
-# Adding this new content with Git or git-annex would be an exercise
-git add sub1/justadir/sub2/anotherdir/afile || true
-
-# DataLad does not require users to determine the correct repository
-# in the tree
-datalad add -d . sub1/justadir/sub2/anotherdir/afile
-
-# Again, all associated changes in the entire dataset tree, up to
-# the given parent dataset, were committed
-git status
-
-# DataLad's 'diff' is able to report the changes from these related
-# commits throughout the repository tree
-datalad diff --revision @~1 -r
diff --git a/notes.txt b/notes.txt
index 7d3dc4c..0483229 100644
--- a/notes.txt
+++ b/notes.txt
@@ -60,3 +60,7 @@ 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 path=/home/me/dl-101/DataLad-101>
[INFO] Applying updates to <Dataset path=/home/me/dl-101/DataLad-101>
update(ok): . (dataset)
```

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
9163dea Merge remote-tracking branch 'refs/remotes/roommate/master'
533ea56 add note about datalad update
d4cce9c Include nesting demo from datalad website
a1d07c5 add note on git annex whereis
a27eb75 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)
```

CHAPTER

TWENTYFIVE

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.

25.1 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.

Part VII

Basics 5 – Tuning datasets to your needs

CHAPTER

TWENTYSIX

DIY CONFIGURATIONS

Back in section *Data safety* (page 107), 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 19) 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[88]. 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.

26.1 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. Where ever 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*[89] 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 store general per-user configurations, such as a default editor[91], 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`

[88] As an alternative to a `git config` command, you could also run configuration templates or procedures (see *Configurations to go* (page 165)) 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.

[89] 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[90].

[90] https://git-scm.com/docs/git-config

[91] 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[92]):

```
nano ~/.gitconfig
```

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"
```

[92] https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/

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 = 0b6bef5c-68c4-465f-8d32-c00ffa64dcfb
        version = 5
        backends = MD5E
[submodule "recordings/longnow"]
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        active = true
[remote "roommate"]
        url = ../mock_user/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
        annex-uuid = 2907ce9f-0de0-4f84-aafd-7c07a8cc3c8a
        annex-ignore = false
```

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[87] 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:

[87] https://git-scm.com/docs/git-config#Documentation/git-config.txt-corefileMode

```
$ cat .git/config
[core]
	repositoryformatversion = 0
	filemode = true
	bare = false
	logallrefupdates = true
	editor = nano
[annex]
	uuid = 0b6bef5c-68c4-465f-8d32-c00ffa64dcfb
	version = 5
	backends = MD5E
[submodule "recordings/longnow"]
	url = https://github.com/datalad-datasets/longnow-podcasts.git
	active = true
[remote "roommate"]
	url = ../mock_user/DataLad-101
	fetch = +refs/heads/*:refs/remotes/roommate/*
	annex-uuid = 2907ce9f-0de0-4f84-aafd-7c07a8cc3c8a
	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` 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.

Find out more: If things go wrong

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 following hidden section can give you a glimpse of this.

Find out more: More on this config file

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 5 repository", and gives the dataset its own git-annex UUID. While the "annex-uuid"[93] 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 111) 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

[93] A UUID is a universally unique identifier – a 128-bit number that unambiguously identifies information.

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 69). 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 61) to see that the "url" is the same URL we cloned the longnow dataset from, and go back to section *Looking without touching* (page 119) 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 139). 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 229). *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.

26.2 `.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 your 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 need to be defined in different files – files that are version controlled. The next section will talk about them.

CHAPTER

TWENTYSEVEN

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 139)). The configuration responsible for this behavior is in a `.gitattributes` file, and we'll start this section by looking into it.

27.1 `.gitattributes`

This file lies right in the root of your superdataset:

```
$ cat .gitattributes

* annex.backend=MD5E
**/.git* annex.largefiles=nothing
* annex.largefiles=(not(mimetype=text/*)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 111) 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 "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=(not(mimetype=text/*))
```

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 111) 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 `(not(mimetype=text/*))`[98], the `text2git` configuration template configured git-annex to regard all files of type text **not** 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`.

Lastly, if you wanted to configure a rule based on **size**, you could add a row such as:

```
** annex.largefiles(largerthan=20kb)
```

[98] 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
```

to store only files exceeding 20KB in git-annex[99].

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 overrule 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[100]. 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[94] 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.

27.2 `.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
```

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

[99] 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 precendence 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.

[100] 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[101] for more information on this command).

[101] https://git-annex.branchable.com/git-annex-migrate/

[94] https://git-annex.branchable.com/tips/largefiles/

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.

Back in section *Looking without touching* (page 119) you may have already seen one additional configuration option in a footnote: The `datalad-recursiveinstall` key. This key is defined on a per subdataset 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.

27.3 `.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[95], 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 = 71f03bec-32ac-11ea-b7a4-e86a64c8054c
```

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.

27.4 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".

[95] http://docs.datalad.org/en/latest/generated/datalad.config.html

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 can read more about this here[96]). 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 1b59b8c..599864e 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,4 +1,4 @@
 [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
```

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
```

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.

27.5 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[102]. Configuration

[96] https://help.github.com/en/articles/which-remote-url-should-i-use

[102] **Some more 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:

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[97] 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) 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`.

27.6 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
```

(continues on next page)

- `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.

[97] https://git-scm.com/book/en/v2/Git-Internals-Environment-Variables

(continued from previous page)

```
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
shared 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)
```

CHAPTER

TWENTYEIGHT

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 pre-configured *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 = '(not(mimetype=text/*))'`, 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 = '(not(mimetype=text/*))'
# check existing configurations:
attrs = ds.repo.get_gitattributes('*')
# if not already an existing configuration, configure git-annex with the above rule
if not attrs.get('*', {}).get(
        'annex.largefiles', None) == annex_largefiles:
    ds.repo.set_gitattributes([
        ('*', {'annex.largefiles': annex_largefiles})])
```

(continues on next page)

(continued from previous page)

```
# this saves and commits the changed .gitattributes file
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. 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_hirni (../../../adina/env/handbook/lib/python3.7/site-packages/datalad_hirni/resources/
↪procedures/cfg_hirni.py) [python_script]
cfg_bids (../../../adina/repos/datalad-neuroimaging/datalad_neuroimaging/resources/
↪procedures/cfg_bids.py) [python_script]
cfg_metadatatypes (../../../adina/repos/datalad/datalad/resources/procedures/cfg_
↪metadatatypes.py) [python_script]
cfg_text2git (../../../adina/repos/datalad/datalad/resources/procedures/cfg_text2git.py)␣
↪[python_script]
cfg_yoda (../../../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[103].

- `cfg_yoda` configures a dataset according to the yoda principles – the section *YODA: Best practices for data analyses in a dataset* (page 179) talks about this in detail.

[103] 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, 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.

- `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.

28.1 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>`

Find out more: 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
```

Find out more: 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 211) will elaborate on this general principle later in the handbook.

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 179)), 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.

Find out more: 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. 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[104]) 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:

[104] 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.

```
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
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"
}
```

(continues on next page)

(continued from previous page)

```
# 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
```

Todo: It might be helpful to have (or reference) a table with all available procedures and a short explanation. Maybe on the cheatsheet.

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 322) 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 299).

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 DataLads procedures"
add(ok): notes.txt (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

CHAPTER

TWENTYNINE

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 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. They can be applied already at creation of a dataset with `datalad create -c <procedure>`, or executed later with a **`datalad run-procedure`** command.

29.1 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.

Part VIII

Basics 6 – Make the most out of datasets

CHAPTER

THIRTY

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.

CHAPTER

THIRTYONE

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.

31.1 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"[105], 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/
```

(continues on next page)

105 http://phdcomics.com/comics/archive.php?comicid=1531

(continued from previous page)

```
│   │   ├── 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 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"[110]. 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 181)
- *P2: Record where you got it from, and where it is now* (page 184)
- *P3: Record what you did to it, and with what* (page 185)

[110] "Why does the acronym contain itself?" you ask confused. "That's because it's a recursive acronym[111], 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."

[111] https://en.wikipedia.org/wiki/Recursive_acronym

As you will see, complying to these principles is easy if you use DataLad. Let's go through them one by one:

31.2 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.
- 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[106] or docker containers[107][112], 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 compre-

[106] https://singularity.lbl.gov/
[107] https://www.docker.com/get-started
[112] 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[113] in the wonderful book The Turing Way[114], a comprehensive guide to reproducible data science, or read about it in section *YODA-compliant data analysis projects* (page 189).
[113] https://the-turing-way.netlify.com/reproducible_environments/06/containers#Containers_section
[114] https://the-turing-way.netlify.com/introduction/introduction

hensive[115] 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
```

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 314) contains a step-by-step instruction on how to build and

[115] This 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[116], to test the functionality of your functions and scripts automatically. If relevant, the setup for continuous integration frameworks (such as Travis[117]) 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 excellent chapter on testing[118] in the book The Turing Way[119].

[116] https://en.wikipedia.org/wiki/Continuous_integration
[117] https://travis-ci.org
[118] https://the-turing-way.netlify.com/testing/testing.html#Acceptance_testing
[119] https://the-turing-way.netlify.com/introduction/introduction

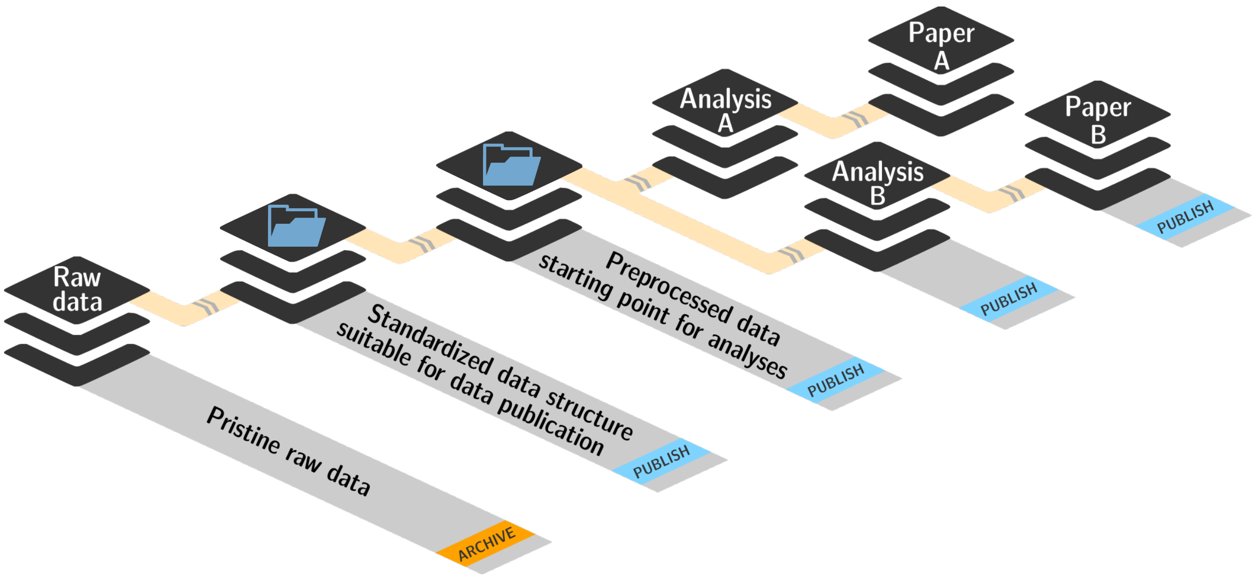

Raw data
Pristine raw data
ARCHIVE
Standardized data structure suitable for data publication
PUBLISH
Preprocessed data starting point for analyses
PUBLISH
Analysis A
Analysis B
PUBLISH
Paper A
Paper B
PUBLISH

Fig. 1: Data are modular components that can be re-used easily.

share such a reproducible paper, if you want to learn more.

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 re-usable 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.

31.3 P2: Record where you got it from, and where it is now

Its good to have data, but its 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 49)) 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.

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 path*s that would only work on your system, but instead as *relative path*s 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[120] 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.

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.

[120] Substitute unfeasible with *wasteful*, *impractical*, or simply *stupid* if preferred.

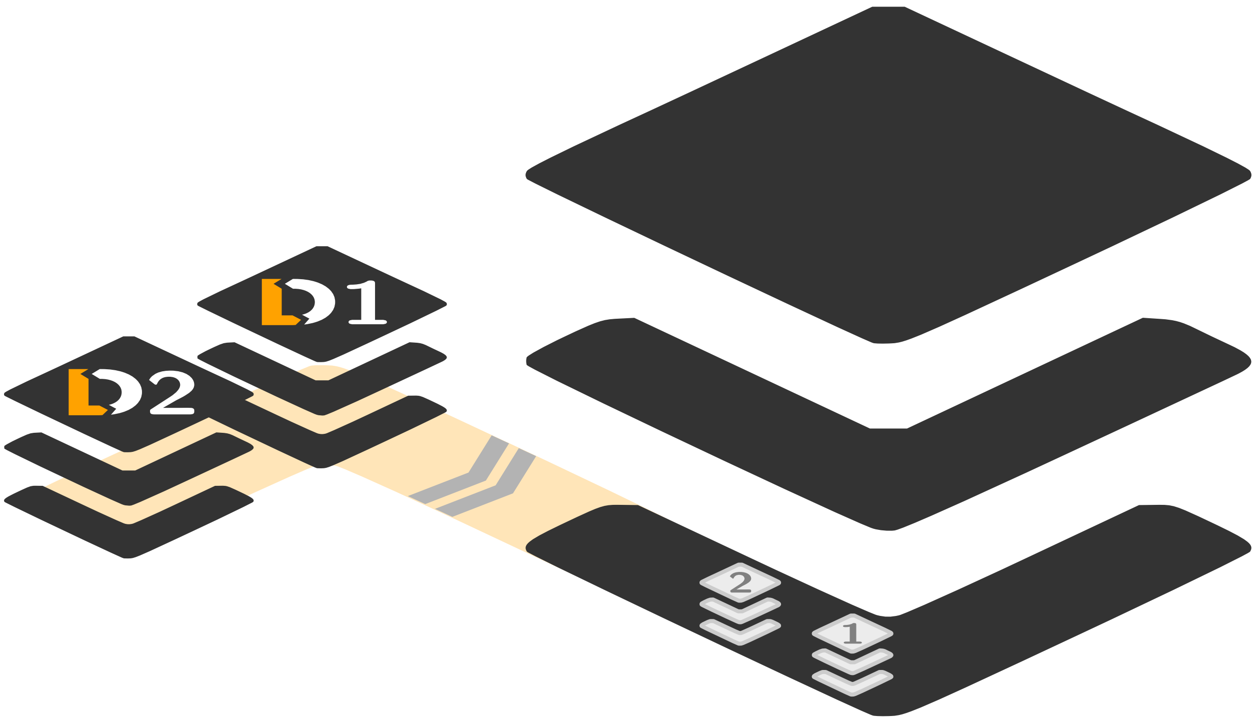


Fig. 2: Schematic illustration of two standalone data datasets installed as subdatasets into an analysis project.

31.4 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 77) already outlined the problem of associating 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 a later part of the book) 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.

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.

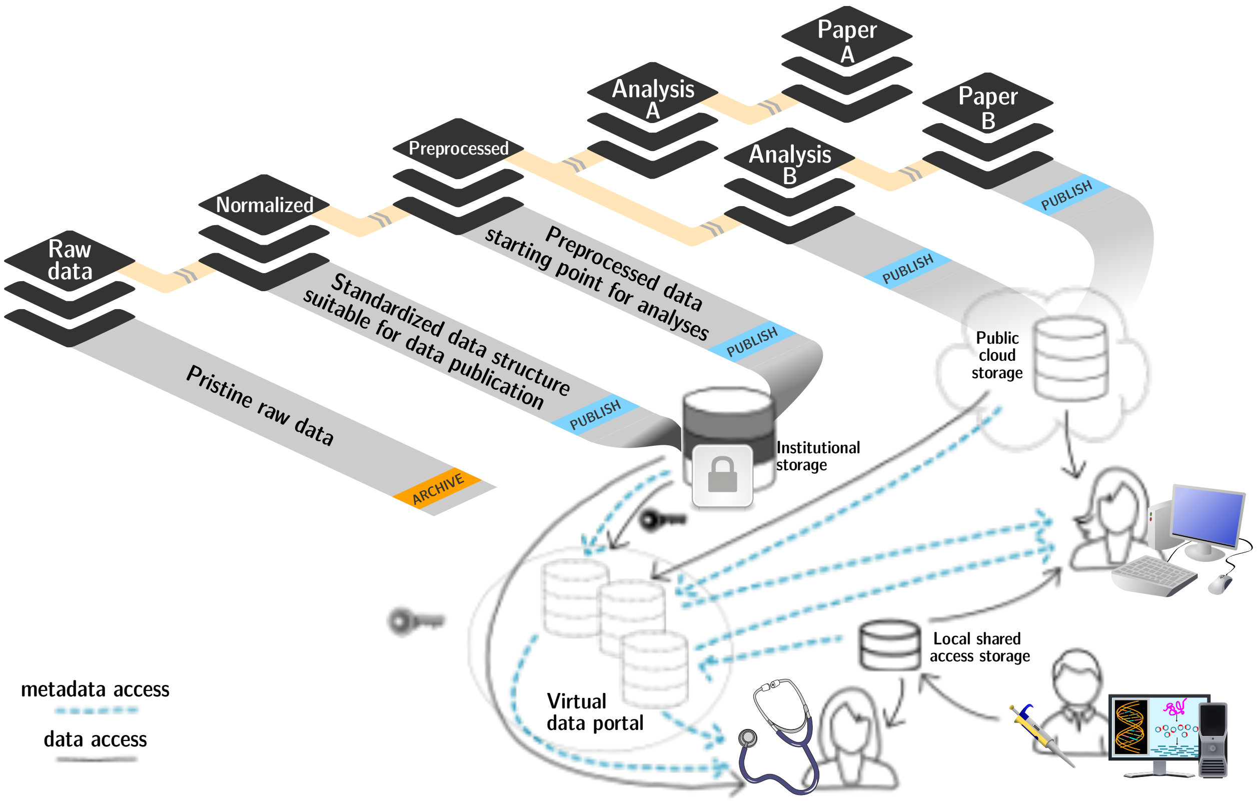


Fig. 3: In a dataset that complies to the YODA principles, modular components (data, analysis results, papers) can be shared or published easily.

Fig. 4: “Feel the force!”

31.5 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 45), 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

$ 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[121]
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.

[121] To re-read how .gitattributes work, go back to section *DIY configurations* (page 151), and to remind yourself about how this worked for the text2git configuration, go back to section *Data safety* (page 107).

31.6 Sources

This section is based on this comprehensive poster[108] and these publicly available slides[109] about the YODA principles.

[108] https://f1000research.com/posters/7-1965
[109] https://github.com/myyoda/talk-principles

CHAPTER

THIRTYTWO

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[122] 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 the hidden section below:

Find out more: 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."

This short section will give you an overview on DataLad's Python API and explore how to make use of it in an analysis project. Together with the previous section on the YODA principles, it is a good basis for a data analysis midterm project in Python.

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[123] 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:

[122] https://www.python.org/
[123] http://docs.datalad.org/en/latest/modref.html

```
>>> from datalad.api import Dataset
```

or like this:

```
>>> import datalad.api as dl
>>> dl.Dataset()
```

A `Dataset` is a class[124] that represents a DataLad dataset. In addition to the stand-alone commands, all of DataLad's functionality is also available via methods[125] 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.

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.

Note: 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 don't 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.

For your midterm project submission, you decide to create a data analysis on the iris flower data set[126]. 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

[124] https://docs.python.org/3/tutorial/classes.html
[125] https://docs.python.org/3/tutorial/classes.html#method-objects
[126] https://en.wikipedia.org/wiki/Iris_flower_data_set

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!

32.1 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 the hidden section below.

Find out more: 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
create(ok): /home/me/dl-101/iris_data (dataset)
```

and subsequently got the data from a publicly available GitHub Gist[127] with a **`datalad download-url`** command:

Find out more: What are GitHub Gists?

GitHub Gists are a particular service offered by GitHub that allow users to share pieces of code snippets and other short/small standalone information. Find out more on Gists here[128].

```
$ cd iris_data
$ datalad download-url https://gist.githubusercontent.com/netj/8836201/raw/
↪6f9306ad21398ea43cba4f7d537619d0e07d5ae3/iris.csv
```

(continues on next page)

[127] https://gist.github.com/netj/8836201
[128] https://help.github.com/en/github/writing-on-github/creating-gists#about-gists

(continued from previous page)

```
[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.

"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."

32.2 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[135]:

```
# 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] 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)
```

(continues on next page)

[135] 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.

(continued from previous page)

```
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 adequately 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`!

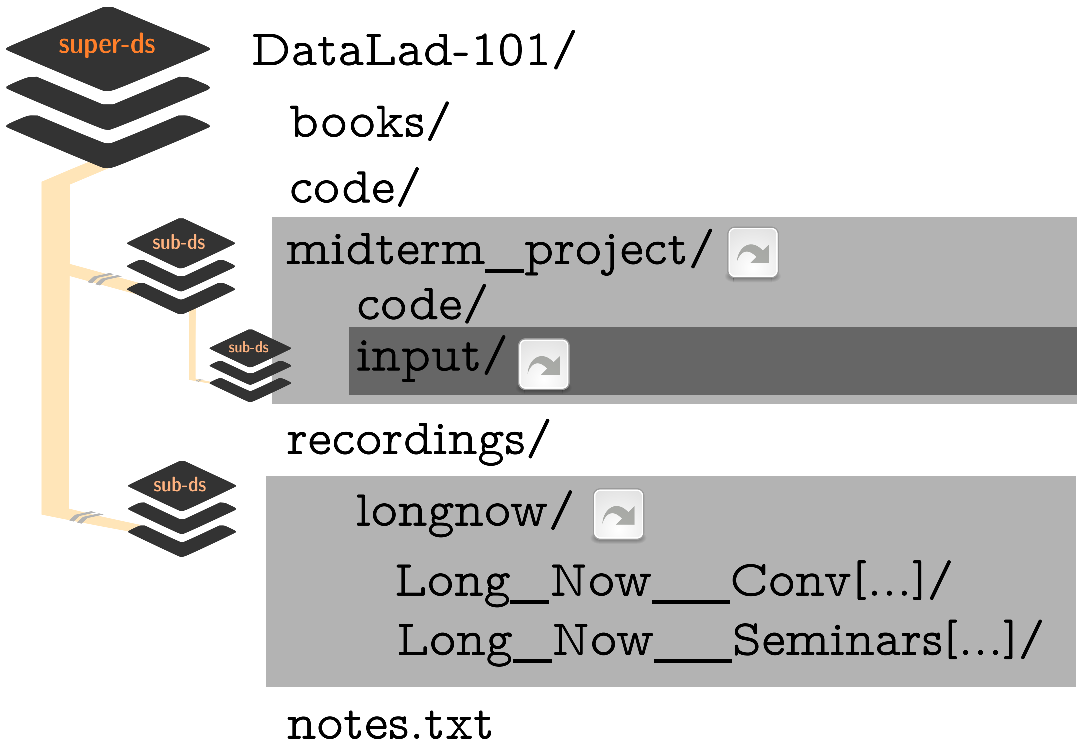
```
$ cd midterm_project
# we are in midterm_project, thus -d . points to the root of it.
$ datalad clone -d . https://github.com/datalad-handbook/iris_data.git input/
[INFO] Cloning https://github.com/datalad-handbook/iris_data.git [1 other candidates] into '/
↪home/me/dl-101/DataLad-101/midterm_project/input'
[INFO]   Remote origin not usable by git-annex; setting annex-ignore
add(ok): input (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
install(ok): input (dataset)
action summary:
  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
```

(continues on next page)



(continued from previous page)

```
│   └── 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:

32.3 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[136]. The iris dataset works well for such questions. Based on the features of the flowers (sepal and petal width and length) you will try to predict what type of flower (*Setosa*,

[136] If you want to know more about this algorithm, this blogpost[137] 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.

[137] https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761

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')
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')
```

(continues on next page)

(continued from previous page)

```
EOT
```

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)
```

Find out more: What is a tag?

*tag*s 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 75ba33af65b84826e8c2007ed586e9edf1caaf3c
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:52:41 2020 +0100

    add script for kNN classification and plotting

diff --git a/code/script.py b/code/script.py
new file mode 100644
index 0000000..e43d58e
```

(continues on next page)

(continued from previous page)

```
--- /dev/null
+++ b/code/script.py
@@ -0,0 +1,41 @@
+
+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')
+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')
+
```

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.

Note: Note that you need to have the following Python packages installed to run the analysis[138]:

- pandas[129]
- seaborn[130]
- sklearn[131]

The packages can be installed via `pip`. Check the footnote[138] 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 with your Python software-setup.

[138] It is recommended (but optional) to create a virtual environment[139] 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
```

[129] https://pandas.pydata.org/
[130] https://seaborn.pydata.org/
[131] https://scikit-learn.org/

```
$ 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"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
get(ok): input/iris.csv (file) [from web...]
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
c8cd941 [DATALAD RUNCMD] analyze iris data with classification analysis
75ba33a add script for kNN classification and plotting
44928da [DATALAD] Recorded changes
d28b8b0 Apply YODA dataset setup
f33e70c [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[140].

```
# 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)
```

[140] 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 151).

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[140].

Find out more: 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*.

Find out more: 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[132] is a similar service with highly similar features, but its source code is free and open, whereas GitHub is a subsidiary of Microsoft.

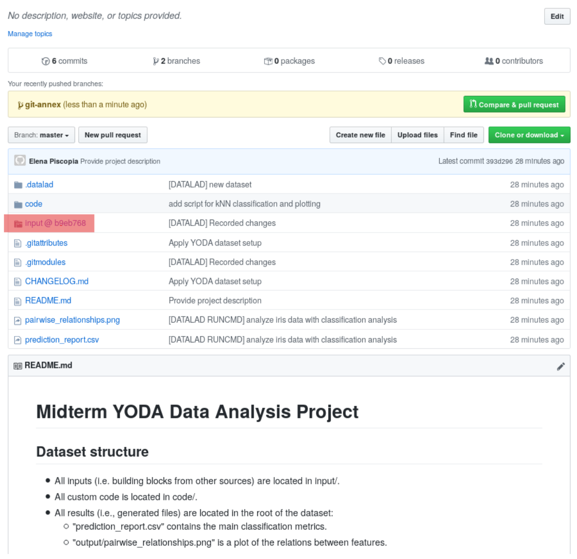
Web-hosting services like GitHub and *GitLab* integrate wonderfully with DataLad. They are especially useful for making your dataset publicly available, if you have figured out storage for your large files otherwise (as large content can not be hosted for free by GitHub). You can make DataLad publish 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.

Note: 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.
- Decide to not follow along.

[132] https://about.gitlab.com/



No description, website, or topics provided.

Your recently pushed branches:

git-annex (less than a minute ago)

Elena Piscopia Provide project description — Latest commit 393d296 28 minutes ago

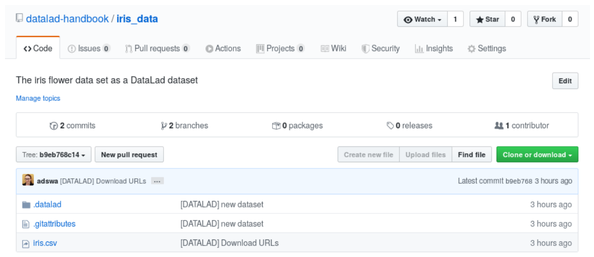
.datalad	[DATALAD] new dataset	28 minutes ago
code	add script for kNN classification and plotting	28 minutes ago
input @ b9eb768	[DATALAD] Recorded changes	28 minutes ago
.gitattributes	Apply YODA dataset setup	28 minutes ago
.gitmodules	[DATALAD] Recorded changes	28 minutes ago
CHANGELOG.md	Apply YODA dataset setup	28 minutes ago
README.md	Provide project description	28 minutes ago
pairwise_relationships.png	[DATALAD RUNCMD] analyze iris data with classification analysis	28 minutes ago
prediction_report.csv	[DATALAD RUNCMD] analyze iris data with classification analysis	28 minutes ago

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.



The iris flower data set as a DataLad dataset

adswa [DATALAD] Download URLs … — Latest commit b9eb768 3 hours ago

.datalad	[DATALAD] new dataset	3 hours ago
.gitattributes	[DATALAD] new dataset	3 hours ago
iris.csv	[DATALAD] Download URLs	3 hours ago

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[133], **datalad create-sibling-gitlab**, `datalad-create-sibling-gitlab` manual).

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 stored in the Git configuration[141], or interactively). 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:

```
$ 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>
```

[133] https://about.gitlab.com/

[141] Such a token can be obtained, for example, using the command line GitHub interface (https://github.com/sociomantic/git-hub) by running: `git hub setup` (if no 2FA is used).

Verify that this worked by listing the siblings of the dataset:

```
$ datalad siblings
.: here(+) [git]
.: github(-) [https://github.com/adswa/midtermproject.git (git)]
```

Note for Git users:

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 publish** 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 publish** (`datalad-publish` manual) command. The **datalad publish** command will make the last saved state of your dataset available to the *sibling* you provide with the `--to` option.

```
$ datalad publish --to github
[INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101/midterm_project> to github
publish(ok): . (dataset) [pushed to github: ['[new branch]', '[new branch]']]
action summary:
  publish (ok: 1)
```

Note for Git users:

The **datalad publish** 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).

Here is one important detail, though: By default, your tags will not be published. 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 to the sibling would be required:

```
$ git push github --tags
```

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!

Find out more: 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[134]:

[134] https://github.com/datalad-handbook/midterm_project

```
$ cd ../../
$ datalad clone "https://github.com/adswa/midtermproject.git"
[INFO] Cloning https://github.com/adswa/midtermproject.git [1 other candidates] into '/home/
↪me/dl-101/midtermproject'
[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 https://github.com/adswa/midtermproject.git/input [3 other candidates] into '/
↪home/me/dl-101/midtermproject/input'
[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
[WARNING] Running get resulted in stderr output: git-annex: get: 2 failed

[ERROR] not available; Try making some of these repositories available:;        71c9c1e7-
↪9ee6-49bc-aa28-ba650cd652c5 -- me@muninn:~/dl-101/DataLad-101/midterm_project; ␣
↪        d1c7b155-731b-405e-b905-47542deff407 -- me@muninn:~/dl-101/DataLad-101/midterm_
↪project;         ef6c82fc-bcc3-46bc-aeb7-fdf93f00e9c9 -- me@muninn:~/dl-101/DataLad-101/
↪midterm_project;         f59223ff-b7bb-4118-821d-351f952c00a3 -- me@muninn:~/dl-101/
↪DataLad-101/midterm_project; ; (Note that these git remotes have annex-ignore set: origin)␣
↪[get(/home/me/dl-101/midtermproject/prediction_report.csv)]
[ERROR] not available; Try making some of these repositories available:;        71c9c1e7-
↪9ee6-49bc-aa28-ba650cd652c5 -- me@muninn:~/dl-101/DataLad-101/midterm_project; ␣
↪        d1c7b155-731b-405e-b905-47542deff407 -- me@muninn:~/dl-101/DataLad-101/midterm_
↪project;         ef6c82fc-bcc3-46bc-aeb7-fdf93f00e9c9 -- me@muninn:~/dl-101/DataLad-101/
↪midterm_project;         f59223ff-b7bb-4118-821d-351f952c00a3 -- me@muninn:~/dl-101/
↪DataLad-101/midterm_project; ; (Note that these git remotes have annex-ignore set: origin)␣
↪[get(/home/me/dl-101/midtermproject/pairwise_relationships.png)]
get(error): prediction_report.csv (file) [not available; Try making some of these␣
↪repositories available:;         71c9c1e7-9ee6-49bc-aa28-ba650cd652c5 -- me@muninn:~/dl-
↪101/DataLad-101/midterm_project;          d1c7b155-731b-405e-b905-47542deff407 --␣
↪me@muninn:~/dl-101/DataLad-101/midterm_project;          ef6c82fc-bcc3-46bc-aeb7-
↪fdf93f00e9c9 -- me@muninn:~/dl-101/DataLad-101/midterm_project;          f59223ff-b7bb-
↪4118-821d-351f952c00a3 -- me@muninn:~/dl-101/DataLad-101/midterm_project; ; (Note that␣
↪these git remotes have annex-ignore set: origin)]
get(error): pairwise_relationships.png (file) [not available; Try making some of these␣
↪repositories available:;         71c9c1e7-9ee6-49bc-aa28-ba650cd652c5 -- me@muninn:~/dl-
↪101/DataLad-101/midterm_project;          d1c7b155-731b-405e-b905-47542deff407 --␣
↪me@muninn:~/dl-101/DataLad-101/midterm_project;          ef6c82fc-bcc3-46bc-aeb7-
↪fdf93f00e9c9 -- me@muninn:~/dl-101/DataLad-101/midterm_project;          f59223ff-b7bb-
↪4118-821d-351f952c00a3 -- me@muninn:~/dl-101/DataLad-101/midterm_project; ; (Note that␣
↪these git remotes have annex-ignore set: origin)]
```

(continues on next page)

(continued from previous page)

```
action summary:
  get (error: 2)
```

Why is that? The file content of these files is managed by git-annex, 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**. A later section

Todo: link 3rd party infra section

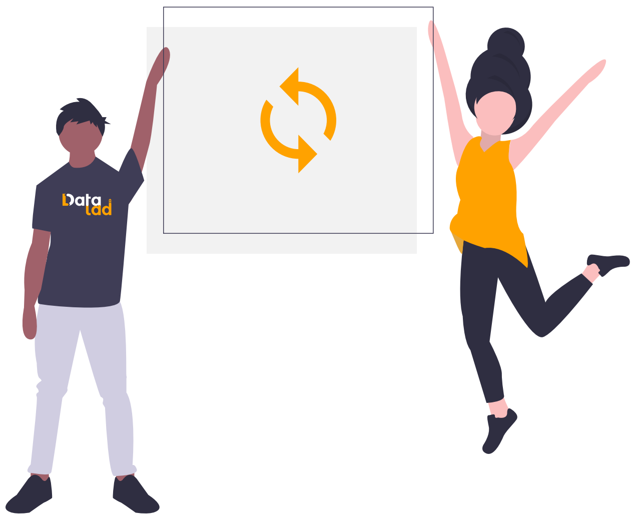
will demonstrate how this can be done. For this dataset, it is not necessary to make the outputs available, though: Because all provenance on their creation was captured, we can simply recompute them with the **datalad rerun** command. If the tag was published we can simply rerun any **datalad run** 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] 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)]
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
remove(ok): pairwise_relationships.png
remove(ok): prediction_report.csv
add(ok): pairwise_relationships.png (file)
add(ok): prediction_report.csv (file)
action summary:
  add (ok: 2)
  get (notneeded: 3)
  remove (ok: 2)
  save (notneeded: 2)
```

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.


DataLad

CHAPTER

THIRTYTHREE

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.

33.1 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.

Part IX

Basics 7 – One step further

CHAPTER

THIRTYFOUR

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
677edbb Provide project description
c8cd941 [DATALAD RUNCMD] analyze iris data with classification analysis
75ba33a add script for kNN classification and plotting
44928da [DATALAD] Recorded changes
d28b8b0 Apply YODA dataset setup
f33e70c [DATALAD] new dataset
```

In principle, this is no news to you. From section *Dataset nesting* (page 69) 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 69) 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 hidden section below.

Find out more: 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 91): 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 229) 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[142].

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)
```

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)
```

Find out more: 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.

[142] http://docs.datalad.org/en/latest/generated/man/datalad-status.html

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 71c53f52a0eed42f82da089e9e352a7ddf36b6ac
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:09 2020 +0100

    finished my midterm project!

diff --git a/midterm_project b/midterm_project
index d28b8b0..677edbb 160000
--- a/midterm_project
+++ b/midterm_project
@@ -1 +1 @@
-Subproject commit d28b8b097c73ab930e407b945da184d724c714a5
+Subproject commit 677edbb02d057865a3cbca678234bf49f987211d
```

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.

CHAPTER

THIRTYFIVE

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[143] extension[155].

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.

[143] http://docs.datalad.org/projects/container/en/latest/
[155] To read more about DataLad’s extensions, see section *DataLad’s extensions* (page 297).

35.1 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[144] 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[156]. An alternative is Singularity[145]. 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. There is Singularity-Hub[146] and Docker-Hub[147]. 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).

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).

[144] https://www.docker.com/

[156] The main reason why Docker is not deployed on HPC systems is because it grants users "superuser privileges[157]". On multi-user systems such as HPC, users should not have those privileges, as it would enable them to temper with other's or shared data and resources, posing a severe security threat.

[157] https://en.wikipedia.org/wiki/Superuser

[145] https://sylabs.io/docs/

[146] https://singularity-hub.org/

[147] https://hub.docker.com/

Note: In order to use Singularity containers (and thus `datalad containers`), you have to install[148] the software singularity.

[148] https://singularity.lbl.gov/docs-installation

35.2 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 77) 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[149], 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:

Find out more: 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[150] 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[151]. 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[152] 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[153] integrates with Github and builds containers from Images pushed to repositories on Github. The docs[154] give you a set of instructions on how to do this.

The **`datalad containers-add`** command takes an arbitrary name to give to the container, and a path or url to a container Image:

[149] https://github.com/datalad-handbook/resources
[150] https://sylabs.io/guides/3.4/user-guide/build_a_container.html
[151] https://github.com/kaczmarj/neurodocker#singularity
[152] https://miykael.github.io/nipype_tutorial/notebooks/introduction_neurodocker.html
[153] https://singularity-hub.org/
[154] https://singularityhub.github.io/singularityhub-docs/

```
# we are in the midterm_project subdataset
$ datalad containers-add midterm-software --url shub://adswa/resources:1
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)
```

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`.

Find out more: What has been added to .datalad/config?

```
$ cat .datalad/config
[datalad "dataset"]
	id = 9efd113c-32ac-11ea-b7a4-e86a64c8054c
[datalad "containers.midterm-software"]
	updateurl = shub://adswa/resources:1
	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.

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 4464d562134fc9a8e4a34344326d86c711f3d72f
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:22 2020 +0100

    [DATALAD] Configure containerized environment 'midterm-software'

diff --git a/.datalad/config b/.datalad/config
index b233709..6bfd89e 100644
--- a/.datalad/config
```

(continues on next page)

(continued from previous page)

```
+++ b/.datalad/config
@@ -1,2 +1,6 @@
 [datalad "dataset"]
        id = 9efd113c-32ac-11ea-b7a4-e86a64c8054c
+[datalad "containers.midterm-software"]
+        updateurl = shub://adswa/resources:1
+        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..800282a
--- /dev/null
+++ b/.datalad/environments/midterm-software/image
@@ -0,0 +1 @@
+../../../.git/annex/objects/zJ/8f/MD5E-s232214559--49dcb6ac1a5787636c9897c4d4df7e10/MD5E-
↪s232214559--49dcb6ac1a5787636c9897c4d4df7e10
\ No newline at end of file
```

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 189) 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" \
  "python3 code/script.py"
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
unlock(ok): pairwise_relationships.png (file)
unlock(ok): prediction_report.csv (file)
add(ok): pairwise_relationships.png (file)
add(ok): prediction_report.csv (file)
save(ok): . (dataset)
action summary:
  add (ok: 2)
  get (notneeded: 4)
```

(continues on next page)

(continued from previous page)

```
  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>
```

Find out more: 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 936952fd146be26992fa172aa127e5a86f93861d
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:28 2020 +0100

    [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": "9efd113c-32ac-11ea-b7a4-e86a64c8054c",
     "exit": 0,
     "extra_inputs": [
      ".datalad/environments/midterm-software/image"
     ],
     "inputs": [
      "input/iris.csv"
     ],
     "outputs": [
      "prediction_report.csv",
      "pairwise_relationships.png"
     ],
     "pwd": "."
    }
```

(continues on next page)

(continued from previous page)

```
    ^^^ Do not change lines above ^^^

diff --git a/pairwise_relationships.png b/pairwise_relationships.png
index 2f69f64..6d00014 120000
--- a/pairwise_relationships.png
+++ b/pairwise_relationships.png
@@ -1 +1 @@
-.git/annex/objects/Pz/Xm/MD5E-s175662--8a9a3e225267f327e1671cf6d01b1957.png/MD5E-s175662--
↪8a9a3e225267f327e1671cf6d01b1957.png
\ No newline at end of file
+.git/annex/objects/z3/23/MD5E-s176597--87d8a72f5f7b1f4f191d0be1bfd15288.png/MD5E-s176597--
↪87d8a72f5f7b1f4f191d0be1bfd15288.png
\ No newline at end of file
diff --git a/prediction_report.csv b/prediction_report.csv
index 42d194b..b46a2d5 120000
--- a/prediction_report.csv
+++ b/prediction_report.csv
@@ -1 +1 @@
-.git/annex/objects/8q/6M/MD5E-s345--a88cab39b1a5ec59ace322225cc88bc9.csv/MD5E-s345--
↪a88cab39b1a5ec59ace322225cc88bc9.csv
\ No newline at end of file
+.git/annex/objects/VF/27/MD5E-s347--7d984f53676358222aa7aa55980f205b.csv/MD5E-s347--
↪7d984f53676358222aa7aa55980f205b.csv
\ 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 reproducible analysis of public neuroimaging data* (page 327).

CHAPTER THIRTYSIX

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 superdataset:

```
$ 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 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.

36.1 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[158] and Docker-Hub[159].

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.

[158] https://singularity-hub.org/
[159] https://hub.docker.com/

Part X

Basics 8 – Help yourself

CHAPTER

THIRTYSEVEN

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.

CHAPTER THIRTYEIGHT

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).

38.1 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 Jan  9 07:53 .
drwxr-xr-x 8 adina adina 4.0K Jan  9 07:53 ..
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 Apr 19  2017 byte-of-python.pdf -> ../.git/annex/objects/F1/Wz/
↪MD5E-s4242644--f4e1c8ebfb5c89a69ff6d268eb2e63e3.pdf/MD5E-s4242644--
↪f4e1c8ebfb5c89a69ff6d268eb2e63e3.pdf
lrwxrwxrwx 1 adina adina  133 Jun 29  2019 progit.pdf -> ../.git/annex/objects/G6/Gj/MD5E-
↪s12465653--05cd7ed561d108c9bcf96022bc78a92c.pdf/MD5E-s12465653--
↪05cd7ed561d108c9bcf96022bc78a92c.pdf
```

(continues on next page)

(continued from previous page)

```
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 2c93e94f6581d655f17e3b223c3b573265396070
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:33 2020 +0100

    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.

Find out more: 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** (which you should have), let's revert the renaming of the the files:

```
$ git reset --hard HEAD~1
$ datalad status
HEAD is now at c2c4282 add container and execute analysis within container
```

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[162], 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 0d83cd6] rename book
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename books/{TLCL.pdf => The_Linux_Command_Line.pdf} (100%)
```

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.

[162] If you want to learn more about the Git-specific concepts of *worktree*, *staging area/index* or *HEAD*, check out section ...

Todo: Write a section on this high-level Git stuff. Maybe in draft of section on Git history...

```
$ git reset --hard HEAD~1
$ datalad status
HEAD is now at c2c4282 add container and execute analysis within container
```

38.2 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 Jan  9 07:53 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 Jan  9 07:53 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.

Find out more: 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 5a617cd7d1e8c901658130c0f57655b66247cc9b
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:36 2020 +0100

    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.

An additional piece of background information: 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!

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.

Finally, let's clean up:

```
$ git reset --hard HEAD~1
HEAD is now at c2c4282 add container and execute analysis within container
```

38.3 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 2cf5ad0c35fdc3c98933c5c660ac5f3bbf3f2dfa
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:37 2020 +0100

    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.

Find out more: Symlinks!

If you have read the additional content in the section *Data integrity* (page 111), 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 Jan  9 07:53 copyofTLCL.pdf -> .git/annex/objects/jf/3M/MD5E-
↪s2120211--06d1efcb05bb2c55cd039dab3fb28455.pdf/MD5E-s2120211--
↪06d1efcb05bb2c55cd039dab3fb28455.pdf
lrwxrwxrwx 1 adina adina 131 Jan  9 07:53 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 242)): 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 c2c4282 add container and execute analysis within container
```

38.4 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 c2c4282 add container and execute analysis within container
```

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
```

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 c2c4282 add container and execute analysis within container
```

```
$ 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**.

Todo: Update this when progress has been made towards https://github.com/datalad/datalad/issues/3464

38.5 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.

Find out more: If a renamed/moved dataset is a sibling...

As section *DIY configurations* (page 151) 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 = 0b6bef5c-68c4-465f-8d32-c00ffa64dcfb
        version = 5
        backends = MD5E
[submodule "recordings/longnow"]
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        active = true
[remote "roommate"]
        url = ../mock_user/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
        annex-uuid = 2907ce9f-0de0-4f84-aafd-7c07a8cc3c8a
        annex-ignore = false
[submodule "midterm_project"]
        url = /home/me/dl-101/DataLad-101/midterm_project
        active = true
[submodule "longnow"]
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        active = true
```

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 path=/home/me/dl-101/DataLad-101>
[ERROR] Cmd('/usr/lib/git-annex.linux/git') failed due to: exit code(128)
|   cmdline: /usr/lib/git-annex.linux/git fetch --progress --prune --recurse-submodules=no -
↪v roommate
|   stderr: '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.' [cmd.py:wait:415] (GitCommandError)
```

Here we go:

```
'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[160] 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 = 0b6bef5c-68c4-465f-8d32-c00ffa64dcfb
        version = 5
        backends = MD5E
[submodule "recordings/longnow"]
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        active = true
[remote "roommate"]
        url = ../mock_user/onemoredir/DataLad-101
        fetch = +refs/heads/*:refs/remotes/roommate/*
```

(continues on next page)

[160] https://en.wikipedia.org/wiki/Sed

(continued from previous page)

```
        annex-uuid = 2907ce9f-0de0-4f84-aafd-7c07a8cc3c8a
        annex-ignore = false
[submodule "midterm_project"]
        url = /home/me/dl-101/DataLad-101/midterm_project
        active = true
[submodule "longnow"]
        url = https://github.com/datalad-datasets/longnow-podcasts.git
        active = true
```

Let's try to update now:

```
$ datalad update
[INFO] Fetching updates for <Dataset path=/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 && git reset --hard master
HEAD is now at c2c4282 add container and execute analysis within container
```

38.6 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 (section *Beyond shared infrastructure* (page 275) will demonstrate how this can be done), 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
```

Your dataset's history records the unannexing of the files.

```
$ git log -p -n 1
commit efb894252c434dca89c158c257d40c3403a73da1
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 10:54:33 2020 +0100

    content removed from git annex

diff --git a/recordings/interval_logo_small.jpg b/recordings/interval_logo_small.jpg
deleted file mode 120000
index f4d6fd6..0000000
--- a/recordings/interval_logo_small.jpg
+++ /dev/null
@@ -1 +0,0 @@
-../.git/annex/objects/36/jF/MD5E-s100877--0fea9537f9fe255d827e4401a7d539e7.jpg/MD5E-s100877-
↪-0fea9537f9fe255d827e4401a7d539e7.jpg
\ No newline at end of file
diff --git a/recordings/salt_logo_small.jpg b/recordings/salt_logo_small.jpg
deleted file mode 120000
index 55ada0f..0000000
--- a/recordings/salt_logo_small.jpg
+++ /dev/null
@@ -1 +0,0 @@
-../.git/annex/objects/xJ/4G/MD5E-s260607--4e695af0f3e8e836fcfc55f815940059.jpg/MD5E-s260607-
↪-4e695af0f3e8e836fcfc55f815940059.jpg
\ No newline at end of file
```

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)
```

38.7 Deleting (annexed) files/directories

Removing files 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.

38.7.1 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  6  2013 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 70c6b87 Added flower mosaic from wikimedia
books
code
```

(continues on next page)

(continued from previous page)

```
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.

38.7.2 Removing annexed content entirely

A different command to remove file content entirely and irreversibly from a repository is the **datalad drop** command (datalad-drop manual). One use case for this is to make a repository more lean. 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 one may want to remove it.

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.

The command will drop file content or directory content from a dataset, but will retain a symlink for this file. 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
[WARNING] Running drop resulted in stderr output: git-annex: drop: 1 failed

```

(continues on next page)

(continued from previous page)

```
[ERROR] unsafe; Could only verify the existence of 0 out of 1 necessary copies; Rather than␣
↪dropping this file, try using: git annex move; (Use --force to override this check, or␣
↪adjust numcopies.) [drop(/home/me/dl-101/DataLad-101/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; Rather than dropping this file, try using: git␣
↪annex move; (Use --force 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 c2c4282 add container and execute analysis within container
```

38.8 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 https://github.com/datalad-datasets/disneyanimation-cloud.git [1 other␣
↪candidates] into '/home/me/dl-101/DataLad-101/cloud'
[INFO]   Remote origin not usable by git-annex; setting annex-ignore
add(ok): cloud (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
install(ok): cloud (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

To uninstall the dataset, use

```
$ datalad uninstall cloud
uninstall(ok): cloud (dataset)
```

(continues on next page)

(continued from previous page)

```
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[163]. 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
# delete the subdataset
$ datalad remove -m "remove obsolete subds" -d . cloud
[INFO] Cloning https://github.com/datalad-datasets/disneyanimation-cloud.git [1 other␣
↪candidates] into '/home/me/dl-101/DataLad-101/cloud'
[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]
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.

Finally, after this last piece of information, let's clean up:

```
$ git reset --hard HEAD~2
HEAD is now at c2c4282 add container and execute analysis within container
```

[163] 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.

38.9 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 111) 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[161] command:

```
chmod -R u+w <dataset>
```

This *recursively* (`-R`, i.e., throughout all files and (sub)directories) gives users (`u`) write permissions (`+w`) for the dataset.

Afterwards, `rm -rf <dataset>` will succeed.

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.

38.10 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.

[161] https://en.wikipedia.org/wiki/Chmod

CHAPTER THIRTYNINE

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[164] 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[168]. Here is an excerpt from the DataLad-101 history to show a few abbreviated hashes of the 15 most recent commits[169]:

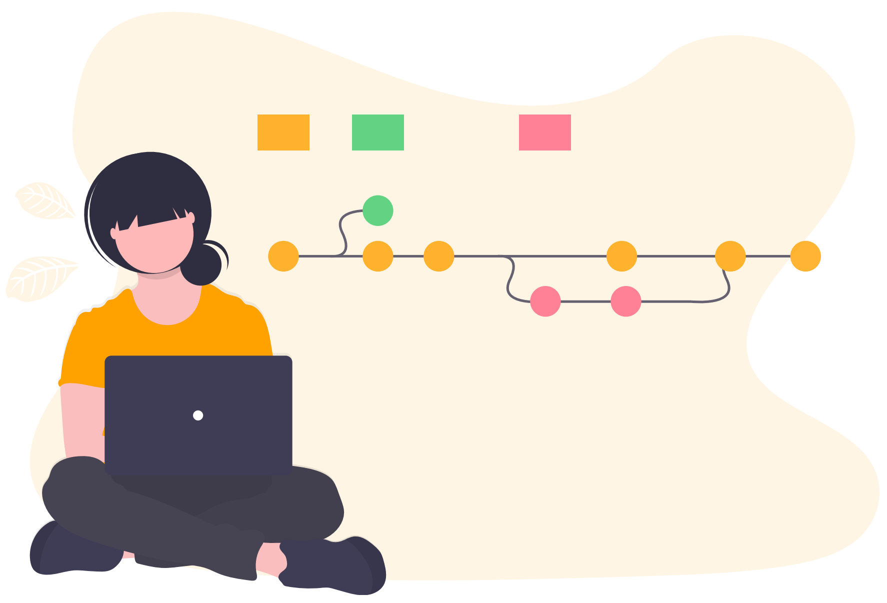
```
$ git log -15 --oneline
c2c4282 add container and execute analysis within container
71c53f5 finished my midterm project!
1449557 [DATALAD] Recorded changes
d765a45 add note on DataLads procedures
097f245 add note on configurations and git config
e55c747 Add note on adding siblings
9163dea Merge remote-tracking branch 'refs/remotes/roommate/master'
533ea56 add note about datalad update
d4cce9c Include nesting demo from datalad website
```

(continues on next page)

[164] https://stackoverflow.com

[168] For example, the **datalad rerun** command introduced in section *DataLad, Re-Run!* (page 83) 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.

[169] 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.



(continued from previous page)

```
a1d07c5 add note on git annex whereis
a27eb75 add note about cloning from paths and recursive datalad get
b64a92b add note on clean datasets
cd4e9c1 [DATALAD RUNCMD] Resize logo for slides
cfd6f24 [DATALAD RUNCMD] Resize logo for slides
3f06057 add additional notes on run options
```

"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!"

39.1 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?
Sorry, 'who' is not a git command - did you mean 'show'?
EOT
```

(continues on next page)

(continued from previous page)

```
$ 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 d2bb1ecd1accb41ab3c7ac9786a6d12b6fcb19c4
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:55 2020 +0100

    [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
```

(continues on next page)

(continued from previous page)

```
--- /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.

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)!

Find out more: Changing the commit messages of not-the-most-recent commits

The **`git commit --amend`** commands 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"[171]. 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.

Note: 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 DataLads 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
```

(continues on next page)

[171] 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.

(continued from previous page)

```
# 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!**[172]

39.2 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.

Important: ote 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!

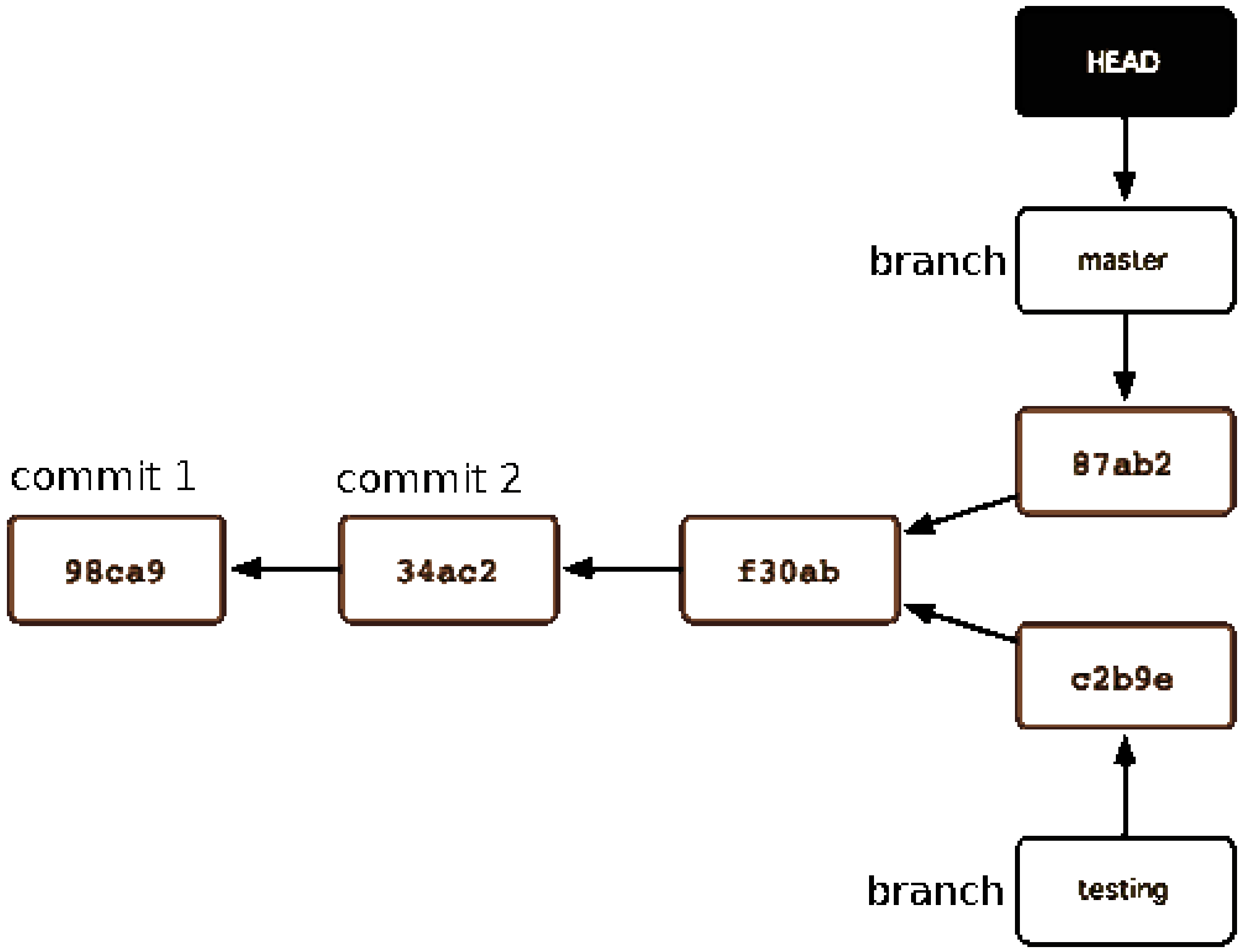
This is a task for the **git reset** command. It essentially allows to undo commits by resetting the history of a dataset to an earlier version. **git reset** comes with several *modes* that determine the exact behavior it, but the relevant one for this aim is `--mixed`[170]. Specifying the command:

```
git reset --mixed COMMIT
```

[172] 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[173].

[173] https://git-scm.com/docs/git-reflog

[170] 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.



will preserve all changes made to files until the specified commit in the dataset, but remove them from the datasets history. This means the commits *until* `COMMIT` (not *including* `COMMIT`) will not be in your history anymore, and instead "untracked files" or "unsaved changes". 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.

Find out more: 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[165] explains the details well.

[165] 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
d2bb1ec [DATALAD] Recorded changes
c2c4282 add container and execute analysis within container
71c53f5 finished my midterm project!
```

```
$ git reset --mixed c2c42825a9f75bb281a5d66f92a6047beb5b9709
```

Let's see what has happened. First, let's check the history:

```
$ git log -n 2 --oneline
c2c4282 add container and execute analysis within container
71c53f5 finished my midterm project!
```

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 29cdf4e0d2aa6575863139b13a381a0194f892de
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:56 2020 +0100

    save my favorite Git joke
```

(continues on next page)

(continued from previous page)

```
commit c2c42825a9f75bb281a5d66f92a6047beb5b9709
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:53:30 2020 +0100

    add container and execute analysis within container
```

Wow! You have rewritten history[171] !

39.3 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)
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 Jan  9 07:53 apdffile.pdf -> .git/annex/objects/fm/4p/MD5E-
↪s1842--62b2153a03cb38eacfe7a1aca718453d.pdf/MD5E-s1842--62b2153a03cb38eacfe7a1aca718453d.
↪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 1842 Jan  9 07:53 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
6035ede [DATALAD] Recorded changes
29cdf4e save my favorite Git joke
c2c4282 add container and execute analysis within container
```

```
$ git reset --mixed 29cdf4e0d2aa6575863139b13a381a0194f892de
```

To see what has happened, let's check the history:

```
$ git log -n 2 --oneline
29cdf4e save my favorite Git joke
c2c4282 add container and execute analysis within container
```

… 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.

39.4 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
29cdf4e save my favorite Git joke
c2c4282 add container and execute analysis within container
71c53f5 finished my midterm project!
1449557 [DATALAD] Recorded changes
d765a45 add note on DataLads procedures
097f245 add note on configurations and git config
e55c747 Add note on adding siblings
```

(continues on next page)

(continued from previous page)

```
9163dea Merge remote-tracking branch 'refs/remotes/roommate/master'
533ea56 add note about datalad update
d4cce9c Include nesting demo from datalad website
a1d07c5 add note on git annex whereis
a27eb75 add note about cloning from paths and recursive datalad get
b64a92b add note on clean datasets
cd4e9c1 [DATALAD RUNCMD] Resize logo for slides
cfd6f24 [DATALAD RUNCMD] Resize logo for slides
3f06057 add additional notes on run options
97f36f1 [DATALAD RUNCMD] convert -resize 450x450 recordings/longn...
b71548b resized picture by hand
4291a9f [DATALAD RUNCMD] convert -resize 400x400 recordings/longn...
a2055bf add note on basic datalad run and datalad rerun
```

Let’s go 15 commits back in time:

```
$ git checkout 97f36f17f90cc92ce4ef4ec1d8629622b26865cd
warning: unable to rmdir 'midterm_project': Directory not empty
Note: switching to '97f36f17f90cc92ce4ef4ec1d8629622b26865cd'.

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 97f36f1 [DATALAD RUNCMD] convert -resize 450x450 recordings/longn...
```

How did your `notes.txt` file look at this point?

```
$ cat notes.txt
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.

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
```

(continues on next page)

(continued from previous page)

```
root of the superdataset with the '-d' option.

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.

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!
```

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 97f36f1 [DATALAD RUNCMD] convert -resize 450x450 recordings/longn.
↪..
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:

```
$ cat notes.txt
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.

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.

There are two useful functions to display changes between two
```

(continues on next page)

(continued from previous page)

```
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.

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!

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.

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.

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>"

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.
```

(continues on next page)

(continued from previous page)

```
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

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
shared 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.

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).
```

… 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**:

```
$ git cat-file --textconv 97f36f17f90cc92ce4ef4ec1d8629622b26865cd:notes.txt
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
```

(continues on next page)

(continued from previous page)

```
(modifications) to history. Note to self:
Always use informative, concise commit messages.

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.

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.

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!
```

The cat-file command is very versatile, and it's documentation[166] 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>
```

39.5 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)
```

(continues on next page)

[166] https://git-scm.com/docs/git-cat-file

(continued from previous page)

```
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
d3a47d2 add joke evaluation to joke
29cdf4e save my favorite Git joke
```

```
$ git reset --hard 29cdf4e0d2aa6575863139b13a381a0194f892de
HEAD is now at 29cdf4e save my favorite Git joke
```

```
$ 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.

39.6 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[172]. 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.

Find out more: Reverting more than a single commit

Alternatively, you can also specify a range of commits modify commits, for example 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 b9e979ac477ed8dbb51db4e36e6f0fdbb828b7e7
[master 26ba583] Revert "did a bad modification"
 Date: Thu Jan 9 07:54:00 2020 +0100
 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 26ba583a128e7727e1f434f5e315ceeff564a369
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:54:00 2020 +0100

    Revert "did a bad modification"

    This reverts commit b9e979ac477ed8dbb51db4e36e6f0fdbb828b7e7.

commit 96e7a1c2f95c7855c81936ace479e55f6c4bd60d
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:54:00 2020 +0100

    add note on helpful git resource

commit b9e979ac477ed8dbb51db4e36e6f0fdbb828b7e7
Author: Elena Piscopia <elena@example.net>
Date:   Thu Jan 9 07:54:00 2020 +0100

    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.

39.7 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[167].

[167] https://help.github.com/en/articles/resolving-a-merge-conflict-using-the-command-line

39.8 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
```

Finally, if you want, apply you're new knowledge about reverting commits to remove the `Gitjoke2.txt` file.

CHAPTER

FORTY

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.

You realize that you already know the most important things: The number one advice on how to get help is "Read the error message."[174]. The second advice it "I'm not kidding: Read the error message[175]. The third advice, finally, says "Honestly, read the f***ing error message[176].

40.1 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[177], 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 0.12.0rc6.dev186
```

If you want a comprehensive overview of your full setup, **`datalad wtf`**[182] 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.

[174] http://poster.keepcalmandposters.com/default/5986752_keep_calm_and_read_the_error_message.png
[175] https://images.app.goo.gl/GWQ82AAJnx1dWtWx6
[176] https://images.app.goo.gl/ddxg4aowbji6XTrw7
[177] https://imgs.xkcd.com/comics/code_quality_3.png
[182] `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.

```
$ datalad wtf
# WTF
## configuration <SENSITIVE, report disabled by configuration>
## datalad
  - version: 0.12.0rc6.dev186
  - full_version: 0.12.0rc6.dev186-g9aa12
## dataset
  - path: /home/me/dl-101/DataLad-101
  - repo: AnnexRepo
  - id: 71f03bec-32ac-11ea-b7a4-e86a64c8054c
  - metadata: <SENSITIVE, report disabled by configuration>
## dependencies
  - cmd:git: 2.20.1
  - tqdm: 4.32.2
  - cmd:annex: 7.20190819+git2-g908476a9b-1~ndall+1
  - cmd:bundled-git: 2.20.1
  - cmd:system-git: 2.24.0
  - cmd:system-ssh: 7.9p1
  - appdirs: 1.4.3
  - boto: 2.49.0
```

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[178]. Alternatively, to get help right inside the terminal, run any command with the `-h`/`--help` option (also shown as an excerpt here):

[178] http://docs.datalad.org/

```
$ datalad get --help
Usage: datalad get [-h] [-s LABEL] [-d PATH] [-r] [-R LEVELS] [-n]
                   [-D DESCRIPTION] [--reckless] [-J NJOBS]
                   [PATH [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 INSTALL 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.

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 . --recursive

*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 [Default: None]

*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
```

(continues on next page)

(continued from previous page)

```
                        another known source. Constraints: value must be a
                        string [Default: None]
  -d PATH, --dataset PATH
                        specify the dataset to perform the add operation on,
                        in which case PATH arguments are interpreted as being
                        relative to this dataset. If no dataset is given, an
                        attempt is made to identify a dataset for each input
                        PATH. Constraints: Value must be a Dataset or a valid
```

This for example is the help page on **datalad get** (the same you would find in the documentation, but in your terminal). 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.

40.2 Asking questions (right)

If nothing you do on your own helps to solve the problem, consider asking others. Check out neurostars[179] and search for your problem – likely, somebody already encountered the same error before[180] 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[181] page prefills a neurostars form with a template for a question for a good starting point if you want to have more guidance or encounter writer's block.

40.3 Common git-annex warnings and errors

A lot of output you will see while working with DataLad originates from git-annex. It's outputs can be wordy and not trivial to comprehend even if everything works. This following section will list some common git-annex warnings and errors and attempts to explain them.

Upon installation of a dataset, you may see:

[179] https://neurostars.org/
[180] http://imgs.xkcd.com/comics/wisdom_of_the_ancients.png
[181] https://github.com/datalad/datalad#support

```
[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
```

This warning lets you know that git-annex will not attempt to download content from the remote "origin", because it is not usable. This can have many reasons, but as long as there are other remotes you can access the data from, you are fine.

Todo: Another type of warning you may encounter during installation is:

```
[INFO   ] Submodule HEAD got detached. Resetting branch master to point to 046713bb.␣
↪Original location was 47e53498
```

Todo: Another one from a missing git config:

```
[WARNING] It is highly recommended to configure git first (set both user.name and user.
↪email) before using DataLad. Failed to verify that git is configured: CommandError:␣
↪command '['git', 'config', 'user.name']' failed with exitcode 1
| Failed to run ['git', 'config', 'user.name'] under None. Exit code=1. out= err= [cmd.
↪py:run:552]CommandError: command '['git', 'config', 'user.email']' failed with exitcode 1
| Failed to run ['git', 'config', 'user.email'] under None. Exit code=1. out= err= [cmd.
↪py:run:552].  Some operations might fail or not perform correctly.
```

Todo: If one does not have an SSH key configured, e.g., on a server (from remodnav paper on brainbfast):

```
[INFO   ] Cloning https://github.com/psychoinformatics-de/paper-remodnav.git/remodnav [2␣
↪other candidates] into '/home/homeGlobal/adina/paper-remodnav/remodnav'
Permission denied (publickey).
[WARNING] Failed to run cmd ['ssh', '-fN', '-o', 'ControlMaster=auto', '-o',
↪'ControlPersist=15m', '-o', 'ControlPath="/home/homeGlobal/adina/.cache/datalad/sockets/
↪6ca483de"', 'git@github.com']. Exit code=255
| stdout: None
| stderr: None
[ERROR  ] Failed to clone from any candidate source URL. Encountered errors per each url␣
↪were: (OrderedDict([('https://github.com/psychoinformatics-de/paper-remodnav.git/remodnav',
↪ "Cmd('/usr/lib/git-annex.linux/git') failed due to: exit code(128)\n  cmdline: /usr/lib/
↪git-annex.linux/git clone --progress -v https://github.com/psychoinformatics-de/paper-
↪remodnav.git/remodnav /home/homeGlobal/adina/paper-remodnav/remodnav [cmd.py:wait:412]"), (
↪'https://github.com/psychoinformatics-de/paper-remodnav.git/remodnav/.git', "Cmd('/usr/lib/
↪git-annex.linux/git') failed due to: exit code(128)\n  cmdline: /usr/lib/git-annex.linux/
↪git clone --progress -v https://github.com/psychoinformatics-de/paper-remodnav.git/
↪remodnav/.git /home/homeGlobal/adina/paper-remodnav/remodnav [cmd.py:wait:412]"), (
↪'git@github.com:psychoinformatics-de/remodnav.git', "Cmd('/usr/lib/git-annex.linux/git')␣
↪failed due to: exit code(128)\n  cmdline: /usr/lib/git-annex.linux/git clone --progress -v␣
↪git@github.com:psychoinformatics-de/remodnav.git /home/homeGlobal/adina/paper-remodnav/
↪remodnav [cmd.py:wait:412]")]),) [install(/home/homeGlobal/adina/paper-remodnav/remodnav)]
[ERROR  ] Installation of subdatasets /home/homeGlobal/adina/paper-remodnav/remodnav failed␣
↪with exception: InstallFailedError:
Failed to install dataset from any of: ['https://github.com/psychoinformatics-de/paper-
↪remodnav.git/remodnav', 'git@github.com:psychoinformatics-de/remodnav.git'] [get.py:_
↪install_subds_from_flexible_source:184] [install(/home/homeGlobal/adina/paper-remodnav/
↪remodnav)]
Traceback (most recent call last):
  File "code/mk_figuresnstats.py", line 811, in <module>
    savefigs(args.figure, args.stats)
  File "code/mk_figuresnstats.py", line 410, in savefigs
    stat)
  File "code/mk_figuresnstats.py", line 274, in confusion
    load_anderson(stimtype, finame)
  File "code/mk_figuresnstats.py", line 28, in load_anderson
    get(fname)
  File "/home/homeGlobal/adina/env/remodnav/lib/python3.5/site-packages/datalad/interface/
↪utils.py", line 492, in eval_func
    return return_func(generator_func)(*args, **kwargs)
  File "/home/homeGlobal/adina/env/remodnav/lib/python3.5/site-packages/datalad/interface/
↪utils.py", line 480, in return_func
    results = list(results)
  File "/home/homeGlobal/adina/env/remodnav/lib/python3.5/site-packages/datalad/interface/
↪utils.py", line 468, in generator_func
    msg="Command did not complete successfully")
datalad.support.exceptions.IncompleteResultsError: Command did not complete successfully [{
↪'type': 'dataset', 'status': 'error', 'action': 'install', 'message': ('Installation of␣
↪subdatasets %s failed with exception: %s', '/home/homeGlobal/adina/paper-remodnav/remodnav
↪', "InstallFailedError: \nFailed to install dataset from any of: ['https://github.com/
↪psychoinformatics-de/paper-remodnav.git/remodnav', 'git@github.com:psychoinformatics-de/
↪remodnav.git'] [get.py:_install_subds_from_flexible_source:184]"), 'path': '/home/
↪homeGlobal/adina/paper-remodnav/remodnav'}]
```

Part XI

Basics 9 Third party infrastructure

CHAPTER

FORTYONE

BEYOND SHARED INFRASTRUCTURE

From the sections *Looking without touching* (page 119) and *YODA-compliant data analysis projects* (page 189) you already know about two common setups for sharing datasets:

Users on a common, shared computational infrastructure such as an *SSH server* can share datasets via simple installations with paths.

Without access to the same computer, it is possible to **publish** datasets to *GitHub* or *GitLab*. You have already done this when you shared your `midterm_project` dataset via *GitHub*. However, this section demonstrated that the files stored in *git-annex* (such as the results of your analysis, `pairwise_comparisons.png` and `prediction_report.csv`) are not published to GitHub: There was meta data about their file availability, but a **datalad get** command on these files failed, because storing (large) annexed content is currently not supported by *GitHub*[207]. In the case of the `midterm_project`, this was not a problem: The computations that you ran were captured with **datalad run**, and others can just recompute your results instead of **datalad get**ting them.

However, not always do two or more parties share the same server, have access to the same systems, or share something that can be recomputed quickly, but need to actually share datasets with data, including the annexed contents.

41.1 Leveraging third party infrastructure

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.

In these cases, the two previous approaches to share data do not suffice. What you would like to do is to provide your friend with a GitHub 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.

To share all dataset contents with your friend, you need to configure an external resource that stores your annexed data contents and that can be accessed by the person you want to share your data with. Such a resource can be a private web server, but also a third party services cloud storage such

[207] *GitLab*, on the other hand, provides a git-annex configuration. 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[208].

[208] https://docs.gitlab.com/ee/administration/git_annex.html

as Dropbox[183], Google[184], Amazon S3 buckets[185], Box.com[186], Figshare[187], owncloud[188], sciebo[189], or many more. The key to achieve this lies within *git-annex*.

In order to use third party services 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.

This tutorial showcases how this can be done, and shows the basics of how datasets can be shared via a third party infrastructure. A much easier alternative using another third party infrastructure is introduced in the next section, *Dataset hosting on GIN* (page 285), using the free G-Node infrastructure. If you prefer this as an easier start, feel free to skip ahead.

From your perspective (as someone who wants to share data), you will need to

- (potentially) install/setup the relevant *special-remote*,
- find a place that large file content can be stored in & set up a *publication dependency* on this location,
- 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 your friend (as someone who wants to obtain a dataset), they will need to

- (potentially) install the relevant *special-remote* 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.

41.2 Setting up 3rd party services to host your data

In this paragraph you will see how a third party service can be configured to host your data. Note that the *exact* procedures are different from service to service – this is inconvenient, but inevitable given the differences between the various third party infrastructures. The general workflow, however, is the same:

1. Implement the appropriate Git-annex *special-remote* (different from service to service).

[183] https://dropbox.com
[184] https://google.com
[185] https://aws.amazon.com/s3/?nc1=h_ls
[186] https://www.box.com/en-gb/home
[187] https://figshare.com/
[188] https://owncloud.org/
[189] https://sciebo.de/

2. Push annexed file content to the third-party service to use it as a storage provider
3. Share the dataset (repository) via GitHub/GitLab/... for others to install from

If the above steps are implemented, others can **install** or **clone** your shared dataset, and **get** or **pull** large file content from the remote, third party storage.

Find out more: 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 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.

As an example, let's walk through all necessary steps to publish `DataLad-101` to **Dropbox**. If you instead are interested in learning how to set up a public Amazon S3 bucket[190], there is a single-page, step-by-step walk-through in the documentation of git-annex[191] that shows how you can create an S3 special remote and share data with anyone who gets a clone of your dataset, without them needing Amazon AWS credentials. Likewise, the documentation provides step-by-step walk-throughs for many other services, such as Google Cloud Storage[192], Box.com[193], Amazon Glacier[194], OwnCloud[195], and many more. Here is the complete list: git-annex.branchable.com/special_remotes/[196].

For Dropbox, the relevant special-remote to configures is rclone[197]. It is a command line program to sync files and directories to and from a large number of commercial providers[209] (Amazon Cloud Drive, Microsoft One Drive, ...). By enabling it as a special remote, *git-annex* gets the ability to do the same, and can thus take care of publishing large file content to such sources conveniently under the hood.

- The first step is to install[198] `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 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.

[190] https://aws.amazon.com/s3/?nc1=h_ls
[191] https://git-annex.branchable.com/tips/public_Amazon_S3_remote/
[192] https://git-annex.branchable.com/tips/using_Google_Cloud_Storage/
[193] https://git-annex.branchable.com/tips/using_box.com_as_a_special_remote/
[194] https://git-annex.branchable.com/tips/using_Amazon_Glacier/
[195] https://git-annex.branchable.com/tips/owncloudannex/
[196] https://git-annex.branchable.com/special_remotes/
[197] https://github.com/DanielDent/git-annex-remote-rclone
[209] `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[210].
[210] http://git-annex.branchable.com/special_remotes/
[198] https://rclone.org/install/

```
$ 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",
```

(continues on next page)

(continued from previous page)

```
        "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, `git clone` the git-annex-remote-rclone[199] repository to your machine (do not clone it into `DataLad-101` but somewhere else on your computer):

  ```
  $ git clone https://github.com/DanielDent/git-annex-remote-rclone.git
  ```

  This is a wrapper around rclone[200] that makes any destination supported by rclone usable with *git-annex*. If you are on a recent version of Debian or Ubuntu, you alternatively can get it more conveniently via your package manager with `sudo apt-get install git-annex-remote-rclone`.

- Copy the path to this repository into your `$PATH` variable. If the clone is in `/home/user-bob/repos`, the command would look like this[211]:

  ```
  $ 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

initremote dropbox-for-friends ok
(recording state in git...)
```

[199] https://github.com/DanielDent/git-annex-remote-rclone

[200] https://rclone.or

[211] 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.

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*:

```
$ datalad siblings
 .: here(+) [git]
 .: dropbox-for-friends(+) [rclone]
 .: roommate(+) [../mock_user/DataLad-101 (git)]
```

On Dropbox, a new folder, `git-annex` will be created for your annexed files. However, this 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[201] for more on this), optionally encrypt content on its way from a local machine to a storage service (see the git-annex documentation[202] 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 command **`datalad-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 189) with the `midterm_project` subdataset). However, in order to be able to link the contents stored in Dropbox, 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 repository (as done in section *YODA-compliant data analysis projects* (page 189)), in order to also publish annex

[201] https://git-annex.branchable.com/chunking/
[202] https://git-annex.branchable.com/encryption/

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.

With this setup, we can publish the dataset to GitHub. Note how the publication dependency is served first:

```
$ datalad publish --to github --transfer-data all
[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`.

The option `--transfer-data` determines how publishing annexed contents should be handled. With the option `all`, *all* annexed contents are published to the third-party data storage. `--transfer-data none`, however, only publishes information stored in Git – that is: The symlink, as information about file availability, but no file content. Anyone who attempts to **`datalad get`** a file from a dataset clone if its contents were not published will fail.

Find out more: What if I don't want to share a dataset with everyone, or only some files of it?

There are a number of ways to restrict access to your dataset or individual files of your dataset. One is via choice of (third party) hosting service for 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`**. On shared file systems you may achieve this via *permissions* for certain groups or users, and for third party infrastructure you may achieve this by invitations/permissions/... options of the respective service.

If it is individual files that you don't 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 don't want to share. This can be done by providing paths to the data that should be published, and the `--transfer-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. The use case

Todo: Write use case "external researcher without data access"

details such a scenario and demonstrates how external collaborators (with whom data can not be shared) can develop scripts against the directory structure and file names of a dataset, submit those scripts to the data owners, and thus still perform an analysis despite not having access to the data.

By publishing only the file contents of `experiment.txt` with

```
$ datalad publish --to github --transfer-data auto 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 **`publish`** will publish the complete dataset history (unless you specify a commit range with the `--since` option – see the manual[203] for more information).

41.3 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[204] `rclone` (as described above).

[203] http://docs.datalad.org/en/latest/generated/man/datalad-publish.html
[204] https://rclone.org/install/

- Run `rclone config` to configure `rclone` to work with Dropbox (as described above). It is important to name the remote "dropbox-for-friends" (i.e., give it the same name as the one configured in the dataset).
- `git clone` the git-annex-remote-rclone[205] repository and copy the path into your `$PATH` variable (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!":

```
$ datalad get books/TLCL.pdf
get(ok): /home/some/other/user/DataLad-101/books/TLCL.pdf (file) [from dropbox-for-friends]
```

41.4 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 build-in support for "exporting" data to other services.

One example is the command **`export-to-figshare`**. Running this command allows you to publish the dataset to Figshare[206]. The main difference is that this moves data out of version control and decentralized tracking, and essentially "throws it over the wall". This means, while your data (also the annexed data) will be available for download on Figshare, none of the special features a DataLad dataset provides will be available, such as its history or configurations.

[205] https://github.com/DanielDent/git-annex-remote-rclone
[206] https://figshare.com/

CHAPTER

FORTYTWO

DATASET HOSTING ON GIN

GIN[212] (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[215].

42.1 Prerequisites

In order to use GIN for hosting and sharing your datasets, you need to

- register
- upload your public *SSH key* for SSH access
- create an empty repository on GIN and publish your dataset to it

Once you have registered[213] 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.

Find out more: 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 *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[214] 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.

[212] https://gin.g-node.org/G-Node/Info/wiki

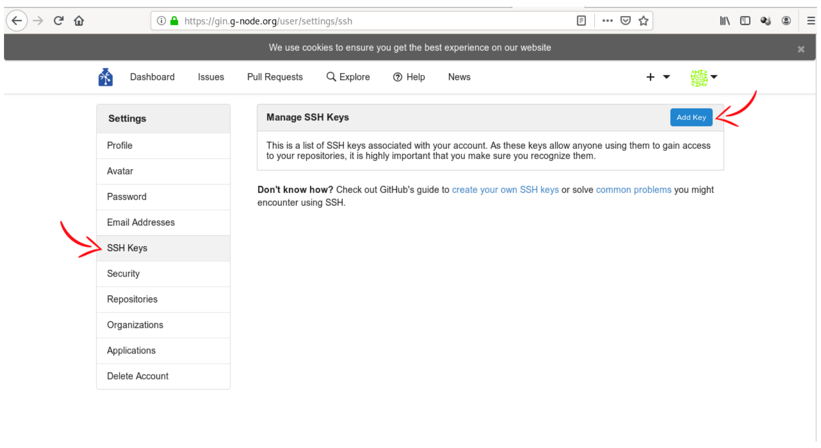
[215] GIN looks and feels similar to GitHub, and among a number advantages, it can assign a *DOI* to your dataset, making it cite-able. Moreover, its web interface[216] and client[217] are useful tools with a variety of features that are worthwhile to check out, as well.

[216] https://gin.g-node.org/G-Node/Info/wiki/WebInterface

[217] https://gin.g-node.org/G-Node/Info/wiki/GinUsageTutorial

[213] https://gin.g-node.org/user/sign_up

[214] https://help.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent



We use cookies to ensure you get the best experience on our website

This is a list of SSH keys associated with your account. As these keys allow anyone using them to gain access to your repositories, it is highly important that you make sure you recognize them.

Don't know how? Check out GitHub's guide to create your own SSH keys or solve common problems you might encounter using SSH.

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[215], 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.

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":

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!

42.2 Publishing your dataset to GIN

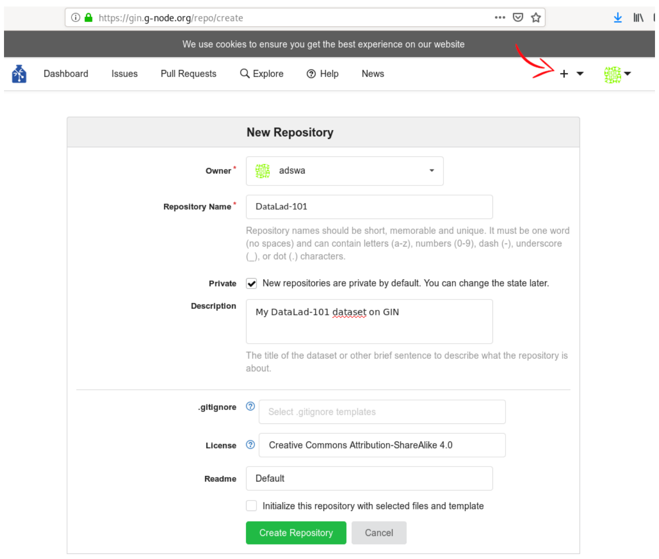
To publish an existing dataset to GIN, create a new, empty repository on GIN first. Unlike with **`datalad create-sibling-github`** (that does this step automatically for you on *GitHub*), this needs to be done via the web interface:

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.
```

(continues on next page)


https://gin.g-node.org/repo/create
We use cookies to ensure you get the best experience on our website
Dashboard
Issues
Pull Requests
Explore
Help
News
+

New Repository

Owner*	adswa
Repository Name*	DataLad-101
	Repository names should be short, memorable and unique. It must be one word (no spaces) and can contain letters (a-z), numbers (0-9), dash (-), underscore (_), or dot (.) characters.
Private	✓ New repositories are private by default. You can change the state later.
Description	My DataLad-101 dataset on GIN
	The title of the dataset or other brief sentence to describe what the repository is about.
.gitignore	Select .gitignore templates
License	Creative Commons Attribution-ShareAlike 4.0
Readme	Default
	☐ Initialize this repository with selected files and template

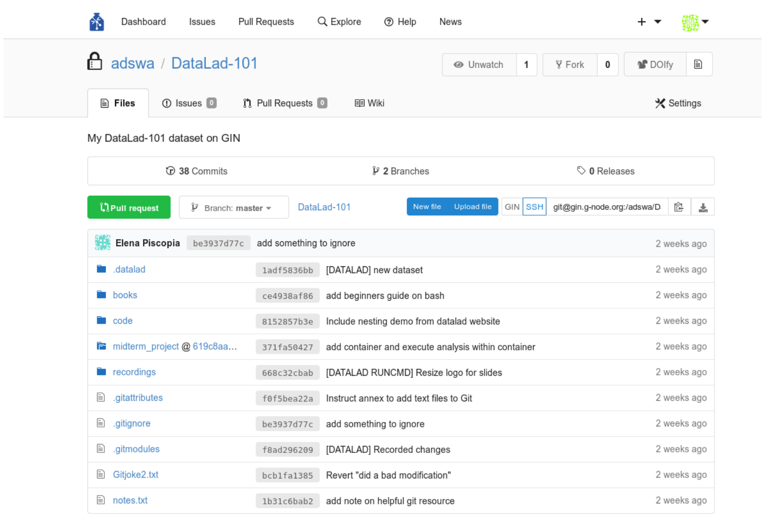
(continued from previous page)

```
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 publish**. As the repository on GIN supports a dataset annex, there is no publication dependency to an external data hosting service necessary, and the dataset contents stored in Git and in git-annex are published to the same place:

```
$ datalad publish --to gin --transfer-data all
 [INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101> data to gin
 publish(ok): books/TLCL.pdf (file)
 publish(ok): books/bash_guide.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 <Dataset path=/home/me/dl-101/DataLad-101> to gin
 Fetching gin (counting objects):  [...]
 publish(ok): . (dataset) [pushed to gin: ['5ea3394..f9a941f', '[new branch]']]
 action summary:
   publish (ok: 7)
```

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.



adswa / DataLad-101

My DataLad-101 dataset on GIN

38 Commits | 2 Branches | 0 Releases

DataLad-101 git@gin.g-node.org:/adswa/D

Elena Piscopia	be3937d77c	add something to ignore	2 weeks ago
.datalad	1adf5836bb	[DATALAD] new dataset	2 weeks ago
books	ce4938af86	add beginners guide on bash	2 weeks ago
code	8152857b3e	Include nesting demo from datalad website	2 weeks ago
midterm_project @ 619c8aa...	371fa50427	add container and execute analysis within container	2 weeks ago
recordings	668c32cbab	[DATALAD RUNCMD] Resize logo for slides	2 weeks ago
.gitattributes	f0f5bea22a	Instruct annex to add text files to Git	2 weeks ago
.gitignore	be3937d77c	add something to ignore	2 weeks ago
.gitmodules	f8ad296209	[DATALAD] Recorded changes	2 weeks ago
Gitjoke2.txt	bcb1fa1385	Revert "did a bad modification"	2 weeks ago
notes.txt	1b31c6bab2	add note on helpful git resource	2 weeks ago

42.3 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`, 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 podcast 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`**[219] (replace the URL shown here with the URL your dataset was published to – likely, you only need to change the user name):

[219] Alternatively, you can configure the siblings url with **`git config`**:

```
$ 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
[submodule "midterm_project"]
        path = midterm_project
        datalad-id = e5a3d370-223d-11ea-af8b-e86a64c8054c
        url = https://github.com/adswa/midtermproject
```

Handily, the **datalad subdatasets** command saved this change to the `.gitmodules` file automatically and the state of the dataset is clean:

```
$ datalad status
```

Afterwards, publish these changes to `gin` and see for yourself how this fixed the problem:

```
$ datalad publish --to gin --transfer-data all
[INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101> data to gin
[INFO   ] Publishing <Dataset path=/home/me/dl-101/DataLad-101> to gin
```

If the subdataset was not published before, you could publish the subdataset to a location of your choice, and modify the `.gitmodules` entry accordingly.

42.4 Sharing the dataset

Once your dataset is published, you can point collaborators and friends to it. 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.

```
$ 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.

CHAPTER

FORTYTHREE

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.

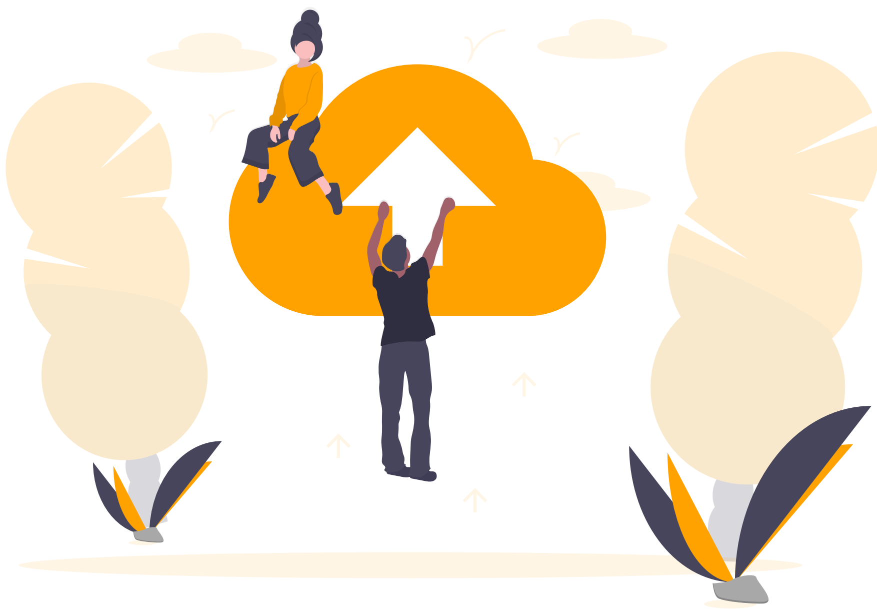
- An easy, free, and fast option is GIN[220], a web-based repository store for scientific data management. If you are registered and have SSH authentication set up, you can create a new, empty repository, add it as a sibling to your dataset, and publish all dataset contents – including annexed data, as GIN supports repositories with an annex.
- Other repository hosting services such as GitHub and GitLab[222]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 275) walked you through how this can be done with Dropbox[221].
- The `--transfer-data` option of **`datalad publish`** allows to either publish `all` or `none` of the annexed contents of your dataset. With `--transfer-data auto` and a path to files, directories, or subdatasets you can publish only selected contents' data.

[220] https://gin.g-node.org

[222] *GitLab* does 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[223].

[223] https://docs.gitlab.com/ee/administration/git_annex.html

[221] https://dropbox.com



43.1 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.

Part XII

Basics 10 – Further options

CHAPTER

FORTYFOUR

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. You know already that this is the reason why a **datalad run** requires a clean dataset: The final **datalad save** after **datalad run** should only save changes that can be attributed to the command that was run, and not changes that existed already but were yet unsaved.

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.

But there are two ways to leave untracked content unaffected by a **datalad save**. One is an option within **datalad save** itself:

```
$ 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[224], 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/*. 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
```

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.

Note: 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.

[224] https://github.com/github/gitignore

CHAPTER

FORTYFIVE

DATALAD'S EXTENSIONS

The commands DataLad provides cover a broad range of domain-agnostic use cases. However, there are extension packages that can add (domain-specific) functionality and new commands.

Such extensions are shipped as separate Python packages, and are *not* included in DataLad itself. Instead, users with the need for a particular extension can install the extension package – either on top of DataLad if DataLad is already installed, or on its own (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.

Among others (a full list can be found on PyPi[225]), the following DataLad extensions are available:

Extension name	Description
DataLad Container[226]	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 215) demonstrates how this extension can be used, as well as the usecase *An automatically reproducible analysis of public neuroimaging data* (page 327).
DataLad Neuroimaging[227]	Metadata extraction support for a range of standards common to neuroimaging data. The usecase *An automatically reproducible analysis of public neuroimaging data* (page 327) demonstrates how this extension can be used.
DataLad Metalad[228]	Equips DataLad with an alternative command suite and advanced tooling for metadata handling (extraction, aggregation, reporting). **Todo:** once section on metadata is done, link it here

To install a DataLad extension, use

[225] https://pypi.org/search/?q=datalad
[226] http://docs.datalad.org/projects/container/en/latest/
[227] https://datalad-neuroimaging.readthedocs.io/en/latest/
[228] http://docs.datalad.org/projects/metalad/en/latest/

```
$ pip install <extension-name>
```

such as in

```
$ pip install datalad-container
```

Afterwards, the new DataLad functionality the extension provides is readily available.

CHAPTER

FORTYSIX

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[229] 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 <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. The following table provides

229 https://en.wikipedia.org/wiki/Hooking

an overview on some of the available keys and their possible values[230]:

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[231] 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:

- 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

[230] The key-value table provides a selection of available key-value pairs, but the 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.

[231] It only needs to be compatible with **`git config`**. This means that it for example should not contain any dots (`.`).

```
{"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 respective 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:

  ```
  $ FIXME 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 operated 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`)[232]:

```
$ 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\"}
```

[232] 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 151). **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.

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.

Part XIII

Use Cases

CHAPTER

FORTYSEVEN

USECASES

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 43) sections, the usecases' code examples are sufficient (though sparser than in Basics) to recreate or apply the solutions they demonstrate.

Use case overview:

47.1 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.

47.1.1 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.

47.1.2 The DataLad Approach

Bob creates his analysis project as a DataLad dataset. Complying with the YODA principles[233], 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.

47.1.3 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] 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:

[233] http://handbook.datalad.org/en/latest/basics/101-123-yoda.html

```
$ 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[234], 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 https://github.com/psychoinformatics-de/studyforrest-data-structural.git [1␣
↪other candidates] into '/home/me/usecases/collab/myanalysis/src/forrest_structural'
[INFO]   Remote origin not usable by git-annex; setting annex-ignore
add(ok): src/forrest_structural (file)
save(ok): . (dataset)
install(ok): src/forrest_structural (dataset)
action summary:
  add (ok: 1)
  install (ok: 1)
  save (ok: 1)
```

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)
```

[234] http://studyforrest.org/

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)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
get(ok): src/forrest_structural/sub-01/anat/sub-01_T1w.nii.gz (file) [from mddatasrc...]
action summary:
  get (notneeded: 1, ok: 1)
  save (notneeded: 2)
```

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 myanalysis into '/home/me/usecases/collab/bobs_analysis'
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 path=/home/me/usecases/collab/myanalysis>
[INFO] Applying updates to <Dataset path=/home/me/usecases/collab/myanalysis>
update(ok): . (dataset)
```

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 publish -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.

47.2 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

Note: 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[235] is an excellent resource.

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

47.2.1 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.

47.2.2 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.

47.2.3 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
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**[237] command will tell him:

[237] If you want to learn more about **git annex whereis**, re-read section *Where's Waldo?* (page 129).

```
$ git annex whereis sources/flowers.jpg
whereis sources/flowers.jpg (2 copies)
          00000000-0000-0000-0000-000000000001 -- web
        f630c8c8-8d1d-4da2-8432-b0a7c5c964dd -- 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[236] 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`**[238] 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)
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`**.

[236] https://imagemagick.org/index.php
[238] If you want to learn more about **`datalad run`**, read on from section *Keeping track* (page 77).

```
$ git log --oneline HEAD~3..HEAD
14d8e8f [DATALAD RUNCMD] extract pimpernel
999b4f5 [DATALAD RUNCMD] extract st-bernard lily
bc38bb3 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
a6e322c [DATALAD RUNCMD] blur image
```

```
$ datalad rerun a6e322c794ef39bea0fb2a077f8eac293d943550
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
unlock(ok): st-bernard-displaced.jpg (file)
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[239].

```
$ git reset --hard HEAD~1
HEAD is now at a6e322c [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] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
[INFO] Making sure inputs are available (this may take some time)
[INFO] == Command start (output follows) =====
[INFO] == Command exit (modification check follows) =====
[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)
add(ok): pimpernel.jpg (file)
save(ok): . (dataset)
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
* 2079cff [DATALAD RUNCMD] blur image
* 9b9ee7b [DATALAD RUNCMD] extract pimpernel
* 92a125b [DATALAD RUNCMD] extract st-bernard lily
| * a6e322c [DATALAD RUNCMD] blur image
```

(continues on next page)

[239] Find out more about working with the history of a dataset with Git in section *Miscellaneous file system operations* (page 229)

(continued from previous page)

```
| * 14d8e8f [DATALAD RUNCMD] extract pimpernel
| * 999b4f5 [DATALAD RUNCMD] extract st-bernard lily
|/
* bc38bb3 Added flower mosaic from wikimedia
* e79b3ff [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
> 2079cff [DATALAD RUNCMD] blur image
= 9b9ee7b [DATALAD RUNCMD] extract pimpernel
= 92a125b [DATALAD RUNCMD] extract st-bernard lily
< a6e322c [DATALAD RUNCMD] blur image
= 14d8e8f [DATALAD RUNCMD] extract pimpernel
= 999b4f5 [DATALAD RUNCMD] extract st-bernard lily
```

Rob can continue processing images, and will turn in a sucessful art project. Long after he finishes high school, he finds his dataset on his old computer again and remembers this small project fondly.

47.3 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[246]. 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.

[246] You can read up on the YODA principles again in section *YODA: Best practices for data analyses in a dataset* (page 179)

47.3.1 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 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.

47.3.2 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 DataLads 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.

Note: The actual manuscript this use case is based on can be found here[240]: 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.

[240] https://github.com/psychoinformatics-de/paper-remodnav/

47.3.3 Step-by-Step

datalad create a DataLad dataset. In this example, it is named "algorithm-paper", and **datalad create** uses the yoda procedure[246] 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[247].

```
$ 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
```

(continues on next page)

[247] You can read up on cloning datasets as subdatasets again in section *Install datasets* (page 61).

(continued from previous page)

```
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 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 path*s instead of hard-coding *absolute path*s.
- 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:

```
# import Datalads 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[248].

```
# 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/`):

[248] Note that the software repository may just as well be cloned into `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 path*s in scripts, data retrieval is automated, and scripts can run on any other computer.

47.3.4 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.

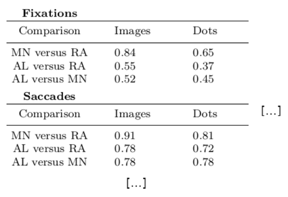
First, if you want to read up on the `\newcommand`, please see its documentation[241]. The command syntax looks like this:

`\newcommand{\name}[num]{definition}`

What we want to do, expressed in the most human-readable form, is this:

`\newcommand{\Table1Cell1Row1}{0.67}`

[241] https://en.wikibooks.org/wiki/LaTeX/Macros



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

[...]

[...]

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):

```
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):

```
\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 \\
    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 ommitted
  \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):

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

```
# 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.

Find out more: 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:

For step 2 and 3, the print statements need to be captured and bound to the `.tex` file. The tee[242] 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.

[242] https://en.wikipedia.org/wiki/Tee_(command)

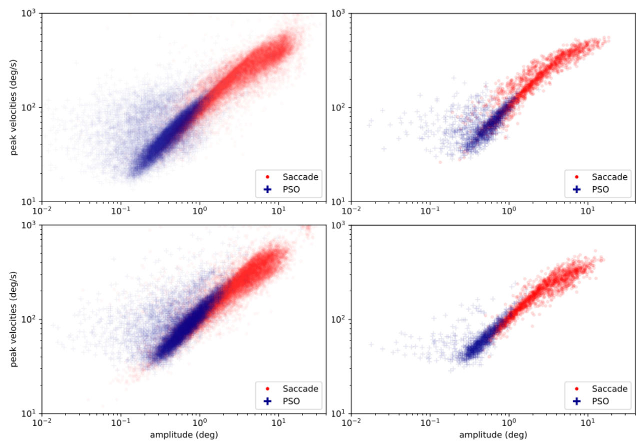


Fig. 6 Main sequence of eye movement events during one 15 minute sequence of the movie (segment 2) for lab (top), and MRI participants (bottom). Data across all participants per dataset is shown on the left, and data for a single exemplary participant on the right.

For step 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[243]. `make` is a decades-old tool known to many and 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:

```
all: main.pdf

main.pdf: main.tex tools.bib EyeGaze.bib results_def.tex figures
    latexmk -pdf -g $<

results_def.tex: code/mk_figuresnstats.py
    bash -c 'set -o pipefail; code/mk_figuresnstats.py -s | tee results_def.tex'

figures: figures-stamp

figures-stamp: code/mk_figuresnstats.py
    code/mk_figuresnstats.py -f -r -m
    $(MAKE) -C img
    touch $@

clean:
    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
    $(MAKE) -C img clean
```

[243] https://en.wikipedia.org/wiki/Make_(software)

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[244]. Any questions can be asked by opening an issue[245].

47.4 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.

[244] http://github.com/psychoinformatics-de/paper-remodnav/
[245] https://github.com/psychoinformatics-de/paper-remodnav/issues/new

47.4.1 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.

47.4.2 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.

47.4.3 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
```

(continues on next page)

(continued from previous page)

```
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[250], 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**[251], **datalad containers-run**[252], and **datalad rerun**[253]. For reference, they also give her the cheat sheet and the link to the DataLad handbook as a resource if she has further questions.

Todo: link cheat sheet once it exists

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[249], and it can easily be added to the dataset with the DataLad-containers extension[252]:

```
$ 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.

[250] Find out more about the YODA principles in section *YODA: Best practices for data analyses in a dataset* (page 179)
[251] Find out more about datalad save in section *Modify content* (page 57)
[252] Find out more about the datalad containers extension in section TODO:link once it exists
[253] Find out more about the datalad rerun command in section *DataLad, Re-Run!* (page 83)
[249] https://singularity-hub.org/

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.

47.5 An automatically reproducible analysis of public neuroimaging data

This use case sketches the basics of an analysis that can be automatically 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 314): 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.

47.5.1 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.

47.5.2 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 clone datasets into an analysis-superdataset makes it easy to share this data together with the analysis. By ensuring that all relevant data are 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.

47.5.3 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] 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[254], and are available from *the DataLad superdataset ///* (datasets.datalad.org). When cloning datasets from this superdataset, we can use its abbreviation `///`. The two datasets, ds000001[255] and ds000002[256], are cloned into the subdirectory `inputs/`.

```
$ cd demo
$ datalad clone -d . ///openfmri/ds000001 inputs/ds000001
[INFO] Cloning http://datasets.datalad.org/openfmri/ds000001 [1 other candidates] into '/
↪home/me/usecases/repro/demo/inputs/ds000001'
add(ok): inputs/ds000001 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
install(ok): inputs/ds000001 (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

```
$ cd demo
$ datalad clone -d . ///openfmri/ds000002 inputs/ds000002
```

(continues on next page)

[254] https://legacy.openfmri.org/
[255] https://legacy.openfmri.org/dataset/ds000001/
[256] https://legacy.openfmri.org/dataset/ds000002/

(continued from previous page)

```
[INFO] Cloning http://datasets.datalad.org/openfmri/ds000002 [1 other candidates] into '/
↪home/me/usecases/repro/demo/inputs/ds000002'
add(ok): inputs/ds000002 (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
install(ok): inputs/ds000002 (dataset)
action summary:
  add (ok: 2)
  install (ok: 1)
  save (ok: 1)
```

Both datasets are now registered as subdatasets, and their precise versions (e.g. in the form of the commit shasum of the lastest 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/
14M	inputs/
```

Both datasets would actually be several gigabytes in size, once the dataset content gets downloaded:

```
$ datalad -C inputs/ds000001 status --annex
$ datalad -C inputs/ds000002 status --annex
130 annex'd files (2.3 GB recorded total size)
274 annex'd files (2.7 GB recorded total size)
```

Both datasets contain brain imaging data, and are compliant with the BIDS standard[257]. This makes it really easy to locate particular images and perform analysis across datasets.

Here we will use a small script that performs 'brain extraction' using FSL[258] 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
```

(continues on next page)

[257] https://bids.neuroimaging.io/
[258] https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/FSL

(continued from previous page)

```
   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).

```
$ 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)
[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) =====
```

(continues on next page)

(continued from previous page)

```
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...]
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)
action summary:
  add (ok: 4)
  get (notneeded: 2, ok: 2)
  save (notneeded: 2, ok: 1)
```

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 5de61bab79e7bb1acd4f1ce988cb83e5266ac99d
Author: Elena Piscopia <elena@example.net>
Date:   Thu Dec 19 09:57:00 2019 +0100

    [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": "78292bba-223d-11ea-af8b-e86a64c8054c",
     "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] Making sure inputs are available (this may take some time)
[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) =====
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)
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 (notneeded: 4, 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): 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): 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
3.1M	.
```

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] Making sure inputs are available (this may take some time)
[INFO] Cloning http://datasets.datalad.org/openfmri/ds000001/.git into '/home/me/usecases/
↪repro/demo/inputs/ds000001'
[INFO] Cloning http://datasets.datalad.org/openfmri/ds000002/.git into '/home/me/usecases/
↪repro/demo/inputs/ds000002'
[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) =====
install(ok): inputs/ds000001 (dataset) [Installed subdataset in order to get /home/me/
↪usecases/repro/demo/inputs/ds000001]
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)
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 (notneeded: 4, 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!

47.6 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 data store so that data are stored in a different place than where people work with it,
2. How to configure the data store and general cluster setup for easy and fast accessibility of data, and
3. How to reduce technical complexities for users and encourage reproducible, version-controlled scientific workflows.

Note: This section is technical in nature and aimed at IT/data management personnel seeking insights into the technical implementation and configuration of a scalable data storage. It is not meant for users of such a data store. A use case about user-facing interactions and workflows with such a data storage is detailed in

Todo: write and link HAMMERPANTS usecase

47.6.1 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 data sets 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. 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, they add all computed derivatives and outputs, even old versions, out of fear of losing work that may become relevant again. This is because version control is not a standard skill in the institute, and especially PhD students and other trainees struggle with the technical overhead of data management *and* data science. Thus, an excess of 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.

47.6.2 The DataLad approach

The compute cluster is refurbished to a state-of-the-art data management system. 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 cluster gets a remote data store: Data lives as DataLad datasets on a different machine than the one the scientific analyses are computed on. For access to the annexed data in datasets, the data store is configured as a git-annex RIA-remote[259]. In case of filesystem inode limitations on the machine serving as the data store (e.g., HPC storage systems), full datasets can be (compressed) 7-zip archives, without losing the ability to query available files. Regardless of the number of file and size of them, such datasets thus use only few inodes. Using DataLad's run-procedures, an institute-wide configuration is distributed among users. Applying the procedure is done in a single **`datalad run-procedure`** command, and users subsequently face minimal technical overhead to interact with the data store.

The infrastructural changes are accompanied by changes in the mindset and workflows of the researchers that perform analyses on the cluster. By using the data store, the institute's work routines are adjusted around DataLad datasets: 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 has the advantage of vastly simplified version control and simplified access to projects for collaborators and supervisors. Data from the data store is cloned as subdatasets. This comes with several benefits: Analyses are automatically linked to data, no unused file copies waste disk space on the compute cluster as data can be retrieved on-demand, and files that are easily re-obtained or recomputed can safely be dropped locally to save even more disk-space. Moreover, upon creation of an analysis project, the associated GitLab project it 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.

47.6.3 Step-by-step

Note: This use case describes the data storage implementation as done in INM-7, research centre Juelich, Germany.

To create a data store, parts of the old compute cluster and parts of the super computer at the Juelich supercomputing centre (JSC) are used to store large amounts of data. Thus, multiple different, independent machines take care of warehousing the data. While this is unconventional, it is convenient: The data does not strain the compute cluster, and with DataLad, it is irrelevant where the data are located.

[259] https://libraries.io/pypi/ria-remote

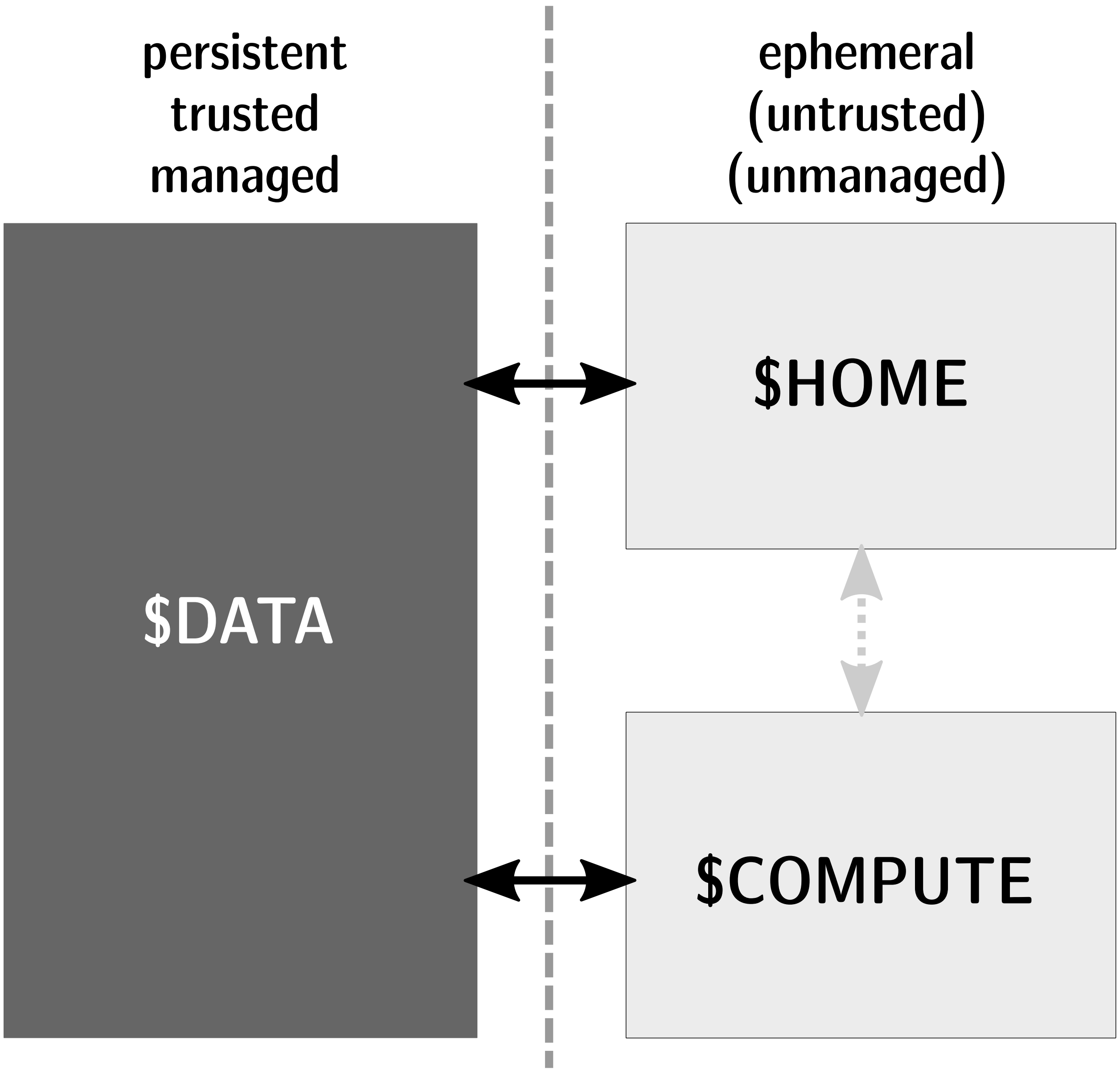


Fig. 1: Trinity of research data handling: The data store (`$DATA`) is managed and backed-up. The compute cluster (`$COMPUTE`) has an analysis-appropriate structure with adequate resources, but just as users workstations/laptops (`$HOME`), it is not concerned with data hosting.

On their own machines (`$HOME`), researchers are free to do whatever they want as long as it is within the limits of their machines. The cluster (`$COMPUTE`) pulls the data exclusively from the data store (`$DATA`). Thus, within `$HOME`, researchers are free to explore data from `$DATA` as they wish, but scaling requires them to use `$COMPUTE`. Results from `$COMPUTE` are pushed back to `$DATA`, and hence anything that is relevant for a computation is tracked (and backed-up) there.

The data store as a git-annex RIA remote

The remote data store exists thanks to git-annex (which DataLad builds upon): Large files in datasets are stored as *values* in git-annex's object tree. A *key* generated from their contents is checked into Git and used to reference the location of the value in the object tree[264]. The object tree (or *keystore*) with the data contents can be located anywhere – its location only needs to be encoded using a *special remote*. This configuration is done on an administrative, system-wide level, and users do not need to care or know about where data are stored, as they can access it just as easily as before.

Find out more: What is a special remote?

A special-remote[260] 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.

The machines in question, parts of an old compute cluster, and parts of the supercomputer at the JSC are configured to receive and store data using the git-annex remote for indexed file archives (RIA[261]) special remote. The git-annex RIA-remote is similar to git-annex's built-in directory[262] special remote, but distinct in certain aspects:

- It allows read access to (compressed) 7z archives, which is a useful feature on systems where light quotas on filesystem inodes are imposed on users, or where one wants to have compression gains. This way, the entire keystore (i.e., all data contents) of the remote that serves as the data store can be put into an archive that uses only a handful of inodes, while remaining fully accessible.
- It provides access to configurable directories via SSH. This makes it easier to accommodate infrastructural changes, especially when dealing with large numbers of repositories, as moving from local to remote operations, or switching target paths can be done by simply changing the configuration.
- It allows a multi-repository directory structure, in which keystore directories of multiple repositories can be organized in to a homogeneous archive directory structure. Importantly, the keystore location in an archive is defined using the **datasets UUID** (in case of DataLad datasets) or the annex remote UUID (in case of any non DataLad dataset). This aids handling

[264] To re-read about how git-annex's object tree works, check out section *Data integrity* (page 111), and pay close attention to the hidden section. Additionally, you can find much background information in git-annex's documentation[265].
[265] https://git-annex.branchable.com/internals/
[260] https://git-annex.branchable.com/special_remotes/
[261] https://libraries.io/pypi/ria-remote
[262] https://git-annex.branchable.com/special_remotes/directory/

of large numbers of repositories in a data store use case, because locations are derived from *repository properties* rather than having to re-configure them explicitly.

The structure under which data is stored in the data store looks like this:

```
082
├── 8ac72-f7c8-11e9-917f-a81e84238a11
│   ├── annex
│   │   ├── objects
│   │   │   ├── ff4
│   │   │   │   └── c57
│   │   │   │       └── MD5E-s4--ba1f2511fc30423bdbb183fe33f3dd0f
│   │   │   ├── abc
│   │   │   │   └── def
│   │   │   │       └── MD5E-s4--ba1f2511fc30423bdbb183fe33f3dd0f
│   │   │   ├── [...]
│   │   └── archives
│   │       └── archive.7z
│   ├── branches
│   ├── config
│   ├── description
│   ├── HEAD
│   ├── hooks
│   │   ├── [...]
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── 04
│   │   │   └── 49b485d128818ff039b4fa88ef57be0cb5b184
│   │   ├── 06
│   │   │   └── 4e5deab57592a54e4e9a495cde70cd6da7605a
│   │   ├── [...]
│   │   ├── info
│   │   └── pack
│   ├── refs
│   │   ├── heads
│   │   │   ├── git-annex
│   │   │   └── master
│   │   └── tags
│   └── ria-layout-version
└── c9d36-f733-11e9-917f-a81e84238a11
    ├── [...]
```

Here is how the RIA-remote features look like in real life:

- Datasets are identified via their *UUID* (e.g., `0828ac72-f7c8-11e9-917f-a81e84238a11`). The UUID is split into the first two levels of the tree structure (as highlighted above in the first two lines), with the two-level structure to avoid exhausting file system limits on the number of files/folders within a directory.
- This first, two-level tree structure can host keystores for any number of repositories.
- The third level holds a directory structure that is identical to a *bare* git repository, and is a clone of the dataset.

Find out more: 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[263].

- Inside of the bare Git repository, the `annex` directory – just as in any standard dataset or repository – contains the keystore (object tree) under `annex/objects` (highlighted above as well). Details on how this object tree is structured are outlined in the hidden section in *Data integrity* (page 111).
- These keystores can be 7zipped if necessary to hold (additional) git-annex objects, either for compression gains, or for use on HPC-systems with inode limitations.

This implementation is fully self-contained, and is a plain file system storage, not a database. Once it is set up, in order to retrieve data from the data store, special remote access to the data store needs to be initialized.

This is done with a custom configuration (`cfg_inm7`) as a run-procedure[266]with a **`datalad create`** command:

```
$ datalad create -c inm7 <PATH>
```

The configuration performs all the relevant setup of the dataset with a fully configured link to `$DATA`: It is configured as a remote to clone and pull data from, but upon creation of the dataset, the dataset's directory is also created at the remote end as a bare repository to enable pushing of results back to `$DATA`. At the same time, a GitLab *sibling* in the institute's GitLab instance is created, with a publication dependency on the data storage.

With this setup, a dataset of any size can be cloned in a matter of seconds by providing its ID as a source in a **`datalad clone`** command:

```
$ datalad clone --dataset mynewdataset \
  --source <ID/URL> \
  mynewdataset/inputs
```

Actual data content can be obtained on demand via **`datalad get`**. Thus, users can selectively obtain only those contents they need instead of having complete copies of datasets as before.

Todo: maybe something about caching here

Upon **`datalad publish`**, computed results can be pushed to the data store and be thus backed-up. Easy-to-reobtain input data can safely be dropped to free disk space on the compute cluster again.

[263] https://git-scm.com/book/en/v2/Git-on-the-Server-Getting-Git-on-a-Server

[266] To re-read about DataLad's run-procedures, check out section *Configurations to go* (page 165). You can find the source code of the procedure on GitLab[267].

[267] https://jugit.fz-juelich.de/inm7/infrastructure/inm7-datalad/blob/master/inm7_datalad/resources/procedures/cfg_inm7.py

With this 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.

Find out more: Software Requirements

- git-annex version 7.20 or newer
- DataLad version 0.12.5 (or later), or any DataLad development version more recent than May 2019 (critical feature: https://github.com/datalad/datalad/pull/3402)
- The `cfg_inm7` run procedure as provided with `pip install git+https://jugit.fz-juelich.de/inm7/infrastructure/inm7-datalad.git`
- Server side: 7z needs to be in the path.

47.7 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 XIV

Appendix

CHAPTER

FORTYEIGHT

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 term) TODO

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.

bash A Unix shell and command language.

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 TODO

clone Git concept: A copy of a *Git* repository. In Git-terminology, all "installed" datasets are clones.

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.

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 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 *clone*s of a dataset and across all its versions.

Debian A common Linux distribution. More information here[268].

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.

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.

GIN A web-based repository store for data management that you can use to host and share datasets. Find out more about GIN here[269].

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[270] or these interactive Git tutorials[271] 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 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.

GitLab An online platform to host and share software projects version controlled with *Git*, similar to *GitHub*. See Gitlab.com[272].

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[273].

master Git concept: The default *branch* in a dataset.

merge Git concept: to integrate the changes of one *branch*/*sibling*/ ... into a different branch.

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

[268] https://www.debian.org/index.en.html
[269] https://gin.g-node.org/G-Node/Info/wiki
[270] https://git-scm.com/book/en/v2
[271] https://try.github.io/
[272] https://about.gitlab.com/
[273] https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Pattern-Matching

picture is, the color depth, the image resolution, when the image was created, the shutter speed, and other data.

nano A common text-editor.

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*.

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[274] 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.

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.

run record A command summary of a **`datalad run`** command, generated by DataLad and included in the commit message.

shasum A hexadecimal number, 40 digits long, that is produced by a secure hash algorithm, and is used by *Git* to identify *commit*s. 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.

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).

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*.

[274] https://pypi.org/

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[275] 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.

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.

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[276]. 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[277].

Ubuntu A common Linux distribution. More information here[278].

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.

[275] https://help.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
[276] http://datasets.datalad.org/
[277] https://jonas.github.io/tig/
[278] https://ubuntu.com

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[279] and how to configure the default editor for git[280].

zsh A Unix shell.

[279] https://www.openvim.com/

[280] https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration

CHAPTER

FORTYNINE

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[281] or a pull request[282].

49.1 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 157) details how these configurations work. This chapter[283] gives a comprehensive overview on what Git is.

49.2 Where is Git's "staging area" in DataLad datasets?

As mentioned in *Populate a dataset* (page 49), a local version control workflow with DataLad "skips" the staging area (that is typical for Git workflows) from the user's point of view.

[281] https://github.com/datalad-handbook/book/issues/new
[282] http://handbook.datalad.org/en/latest/contributing.html
[283] https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

49.3 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 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 *Data integrity* (page 111) can give you more insights into how git-annex takes care of your data. git-annex's website[284] can give you a complete walk-through and detailed technical background information.

49.4 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`.
- **Todo:** more here.

284 https://git-annex.branchable.com/

49.5 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[285], 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.

Todo: sketch a typical workflow, link a yet to write section on sharing on third party infrastructure

49.6 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.

49.7 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.

49.8 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 151) and *More on DIY configurations* (page 157) for additional insights on this).

[285] https://www.dropbox.com/

49.9 How can I cite DataLad?

There is no official paper on DataLad (yet). To cite it, please use the latest zenodo[286] entry found here: https://zenodo.org/record/3512712.

49.10 What is the difference between DataLad, Git LFS, and Flywheel?

Flywheel[287] is an informatics platform for biomedical research and collaboration.

Git Large File Storage[288] (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.

Todo: this.

A more elaborate delineation from related solutions can be found in the DataLad developer documentation[289].

49.11 DataLad version-controls my large files – great. But how much is saved in total?

Todo: this.

49.12 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 111) 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`[293], it is sufficient to use the `-L/--dereference` option. This will follow symbolic links, and make sure that content gets moved instead of symlinks. 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.

[286] https://zenodo.org
[287] https://flywheel.io/
[288] https://github.com/git-lfs/git-lfs
[289] http://docs.datalad.org/en/latest/related.html
[293] The absolutely amazing Midnight Commander[294] `mc` can also follow symlinks.
[294] https://github.com/MidnightCommander/mc

49.13 Is there Python 2 support for DataLad?

No, Python 2 support has been dropped in September 2019[290].

49.14 Is there a graphical user interface for DataLad?

No, DataLad's functionality is available in the command line or via it's Python API.

Todo: maybe update this by mentioning the DataLad webapp extension

49.15 How does DataLad interface with OpenNeuro?

OpenNeuro[291] 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 at github.com/OpenNeuroDatasets[292]. You can obtain the datasets just as any other DataLad datasets with **`datalad clone`** or **`datalad install`**.

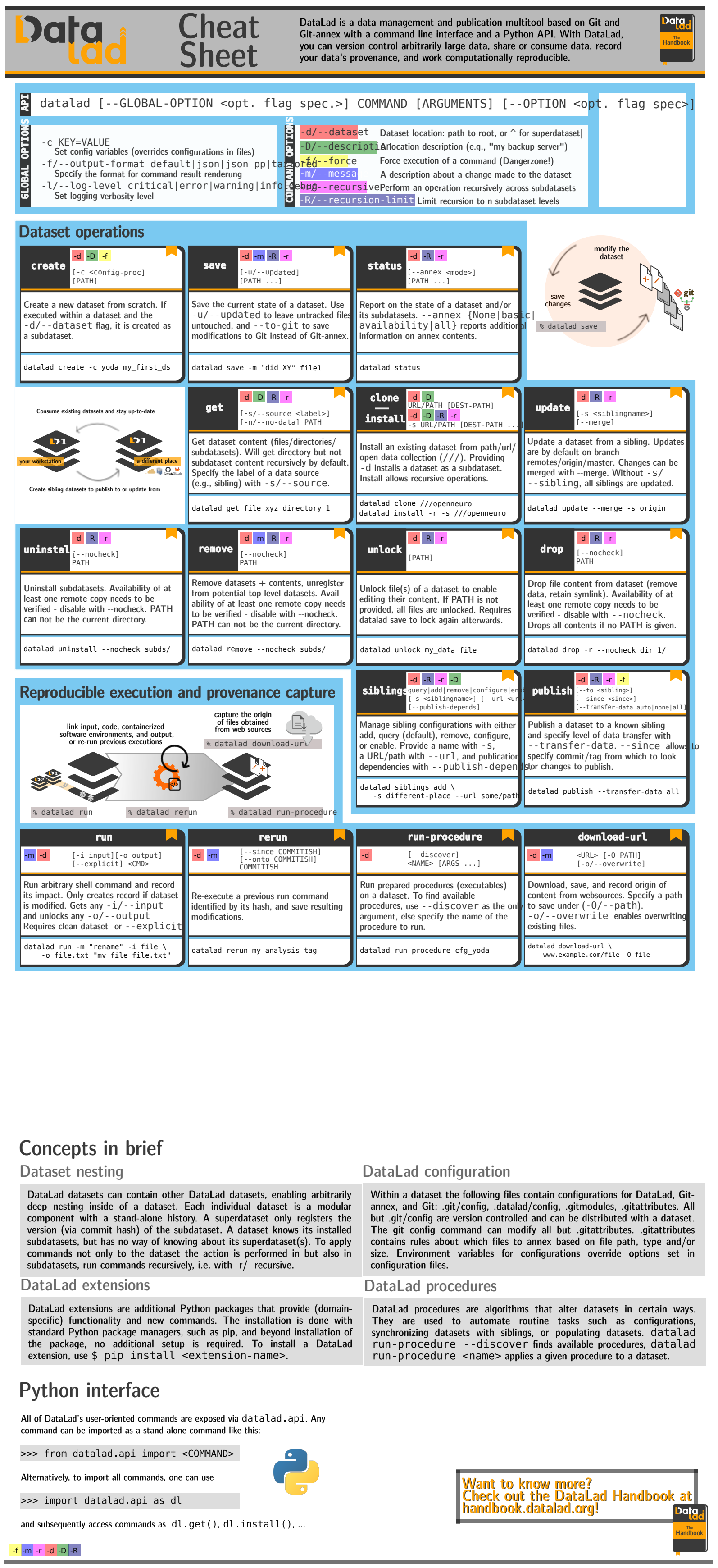
[290] https://github.com/datalad/datalad/pull/3629
[291] https://openneuro.org/
[292] https://github.com/OpenNeuroDatasets

CHAPTER FIFTY

DATALAD CHEAT SHEET

Click on the image below to obtain a PDF version of the cheat sheet. Individual sections are linked to chapters or technical docs.

295 https://github.com/datalad-handbook/artwork/blob/master/src/datalad-cheatsheet.pdf



Cheat Sheet

DataLad is a data management and publication multitool based on Git and Git-annex with a command line interface and a Python API. With DataLad, you can version control arbitrarily large data, share or consume data, record your data's provenance, and work computationally reproducible.

API

```
datalad [--GLOBAL-OPTION <opt. flag spec.>] COMMAND [ARGUMENTS] [--OPTION <opt. flag spec>]
```

GLOBAL OPTIONS

`-c KEY=VALUE`
Set config variables (overrides configurations in files)

`-f/--output-format default|json|json_pp|tailored`
Specify the format for command result renderung

`-l/--log-level critical|error|warning|info|debug`
Set logging verbosity level

COMMAND OPTIONS

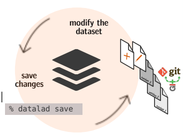
`-d/--dataset`	Dataset location: path to root, or ^ for superdataset
`-D/--description`	Allocation description (e.g., "my backup server")
`-f/--force`	Force execution of a command (Dangerzone!)
`-m/--message`	A description about a change made to the dataset
`-r/--recursive`	Perform an operation recursively across subdatasets
`-R/--recursion-limit`	Limit recursion to n subdataset levels

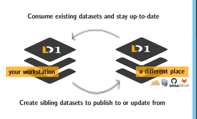
Dataset operations

create -d -D -f
`[-c <config-proc>] [PATH]`
Create a new dataset from scratch. If executed within a dataset and the `-d/--dataset` flag, it is created as a subdataset.
`datalad create -c yoda my_first_ds`

save -d -m -R -r
`[-u/--updated] [PATH ...]`
Save the current state of a dataset. Use `-u/--updated` to leave untracked files untouched, and `--to-git` to save modifications to Git instead of Git-annex.
`datalad save -m "did XY" file1`

status -d -R -r
`[--annex <mode>] [PATH ...]`
Report on the state of a dataset and/or its subdatasets. `--annex {None|basic|availability|all}` reports additional information on annex contents.
`datalad status`




Consume existing datasets and stay up-to-date
your workstation
a different place
Create sibling datasets to publish to or update from

get -d -D -R -r
`[-s/--source <label>] [-n/--no-data] PATH`
Get dataset content (files/directories/subdatasets). Will get directory but not subdataset content recursively by default. Specify the label of a data source (e.g., sibling) with `-s/--source`.
`datalad get file_xyz directory_1`

clone — install -d -D `URL/PATH [DEST-PATH]` -d -D -R -r `-s URL/PATH [DEST-PATH ...]`
Install an existing dataset from path/url/open data collection (///). Providing `-d` installs a dataset as a subdataset. Install allows recursive operations.
```
datalad clone ///openneuro
datalad install -r -s ///openneuro
```

update -d -R -r
`[-s <siblingname>] [--merge]`
Update a dataset from a sibling. Updates are by default on branch remotes/origin/master. Changes can be merged with --merge. Without `-s/--sibling`, all siblings are updated.
`datalad update --merge -s origin`

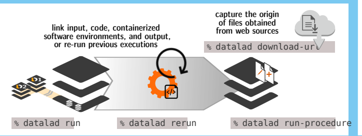
uninstall -d -R -r
`[--nocheck] PATH`
Uninstall subdatasets. Availability of at least one remote copy needs to be verified - disable with --nocheck. PATH can not be the current directory.
`datalad uninstall --nocheck subds/`

remove -d -m -R -r
`[--nocheck] PATH`
Remove datasets + contents, unregister from potential top-level datasets. Availability of at least one remote copy needs to be verified - disable with --nocheck. PATH can not be the current directory.
`datalad remove --nocheck subds/`

unlock -d -R -r
`[PATH]`
Unlock file(s) of a dataset to enable editing their content. If PATH is not provided, all files are unlocked. Requires datalad save to lock again afterwards.
`datalad unlock my_data_file`

drop -d -R -r
`[--nocheck] PATH`
Drop file content from dataset (remove data, retain symlink). Availability of at least one remote copy needs to be verified - disable with `--nocheck`. Drops all contents if no PATH is given.
`datalad drop -r --nocheck dir_1/`

Reproducible execution and provenance capture


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

siblings -d -R -r -D
`query|add|remove|configure|enable [-s <siblingname>] [--url <url>] [--publish-depends]`
Manage sibling configurations with either add, query (default), remove, configure, or enable. Provide a name with `-s`, a URL/path with `--url`, and publication dependencies with `--publish-depends`.
```
datalad siblings add \
  -s different-place --url some/path
```

publish -d -R -r -f
`[--to <sibling>] [--since <since>] [--transfer-data auto|none|all]`
Publish a dataset to a known sibling and specify level of data-transfer with `--transfer-data`. `--since` allows to specify commit/tag from which to look for changes to publish.
`datalad publish --transfer-data all`

run -m -d
`[-i input][-o output] [--explicit] <CMD>`
Run arbitrary shell command and record its impact. Only creates record if dataset is modified. Gets any `-i/--input` and unlocks any `-o/--output` Requires clean dataset or `--explicit`
```
datalad run -m "rename" -i file \
  -o file.txt "mv file file.txt"
```

rerun -d -m
`[--since COMMITISH] [--onto COMMITISH] COMMITISH`
Re-execute a previous run command identified by its hash, and save resulting modifications.
`datalad rerun my-analysis-tag`

run-procedure -d
`[--discover] <NAME> [ARGS ...]`
Run prepared procedures (executables) on a dataset. To find available procedures, use `--discover` as the only argument, else specify the name of the procedure to run.
`datalad run-procedure cfg_yoda`

download-url -d -m
`<URL> [-O PATH] [-o/--overwrite]`
Download, save, and record origin of content from websources. Specify a path to save under (`-O/--path`). `-o/--overwrite` enables overwriting existing files.
```
datalad download-url \
  www.example.com/file -O file
```

Concepts in brief

Dataset nesting

DataLad datasets can contain other DataLad datasets, enabling arbitrarily deep nesting inside of a dataset. Each individual dataset is a modular component with a stand-alone history. A superdataset only registers the version (via commit hash) of the subdataset. A dataset knows its installed subdatasets, but has no way of knowing about its superdataset(s). To apply commands not only to the dataset the action is performed in but also in subdatasets, run commands recursively, i.e. with -r/--recursive.

DataLad configuration

Within a dataset the following files contain configurations for DataLad, Git-annex, and Git: .git/config, .datalad/config, .gitmodules, .gitattributes. All but .git/config are version controlled and can be distributed with a dataset. The git config command can modify all but .gitattributes. .gitattributes contains rules about which files to annex based on file path, type and/or size. Environment variables for configurations override options set in configuration files.

DataLad extensions

DataLad extensions are additional Python packages that provide (domain-specific) functionality and new commands. The installation is done with standard Python package managers, such as pip, and beyond installation of the package, no additional setup is required. To install a DataLad extension, use `$ pip install <extension-name>`.

DataLad procedures

DataLad procedures are algorithms that alter datasets in certain ways. They are used to automate routine tasks such as configurations, synchronizing datasets with siblings, or populating datasets. `datalad run-procedure --discover` finds available procedures, `datalad run-procedure <name>` applies a given procedure to a dataset.

Python interface

All of DataLad's user-oriented commands are exposed via `datalad.api`. 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.install()`, ...

-f -m -r -d -D -R

Want to know more?
Check out the DataLad Handbook at handbook.datalad.org!

295

CHAPTER

FIFTYONE

CONTRIBUTING

Thanks for being curious about contributing! We greatly appreciate and welcome contributions to this book, be it in the form of an issue[296], 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 362) 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.

51.1 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 360). 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

[296] https://github.com/datalad-handbook/book/issues/new

```
$ datalad install -r https://github.com/datalad-handbook/book.git
```

- optional, but recommended: Create a virtual environment

```
$ 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 358) 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[297], 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` in the root of the repository, and open it in your browser, for example with `firefox docs/_build/html/index.html`.

51.2 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

[297] https://en.wikipedia.org/wiki/Elena_Cornaro_Piscopia

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 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` 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.

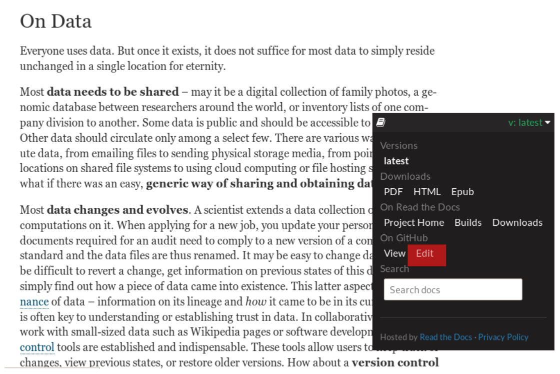
Other custom directives: Other custom directives are `gitusernote` (for additional Git-related information for Git-users), and `findoutmore` (foldable sections that contain content that goes beyond the basics). Make use of them, if applicable to your contribution.

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[298] 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
  ```

[298] https://github.com/datalad/datalad/blob/master/tools/cast_live



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 needs 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 is public and should be accessible to
Other data should circulate only among a select few. There are various wa
ute data, from emailing files to sending physical storage media, from poi
locations on shared file systems to using cloud computing or file hosting s
what if there was an easy, **generic way of sharing and obtaining dat**

Most **data changes and evolves**. A scientist extends a data collection o
computations on it. When applying for a new job, you update your person
documents required for an audit need to comply to a new version of a com
standard and the data files are thus renamed. It may be easy to change da
be difficult to revert a change, get information on previous states of this d
simply find out how a piece of data came into existence. This latter aspect
nance of data – information on its lineage and *how* it came to be in its cu
is often key to understanding or establishing trust in data. In collaborativ
work with small-sized data such as Wikipedia pages or software developm
control tools are established and indispensable. These tools allow users to
changes, view previous states, or restore older versions. How about a **version control**

Fig. 1: You can find an easy way to submit a pull request right from within the handbook.

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)
   :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[299] to execute them:

```
~ course$ tools/cast_live path/to/casts
```

The section *Teaching with the DataLad Handbook* (page 363) outlines more on this and other teaching materials the handbook provides.

51.3 Easy pull requests

The easiest way to do a pull request is within the web-interface that GitHub and readthedocs[300] provide. If you visit the rendered version of the handbook at handbook.datalad.org[301] 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.

But you of course are also welcome to submit a pull request with whichever other workflow suites you best.

[299] https://github.com/datalad-handbook/course
[300] https://readthedocs.org
[301] http://handbook.datalad.org/

51.4 Desired structure of the book

The book consists of three major parts: Introduction, 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.

51.4.1 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.

51.4.2 Basics

- This part contains hands-on-style content on skills that are crucial for using DataLad productively. Any non-essential information is not in basics, but collected in an appendix.
- 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").

51.4.3 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.

51.5 Acknowledging Contributors

If you have helped this project, we would like to acknowledge your contribution in the GitHub repository[302] in our README with allcontributors.org[303], and the project's .zenodo[304] and CONTRIBUTORS.md[305] files. The allcontributors bot[306] will give credit for various types of contributions[307]. 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.

[302] https://github.com/datalad-handbook/book
[303] https://allcontributors.org/
[304] https://github.com/datalad-handbook/book/blob/master/.zenodo.json
[305] https://github.com/datalad-handbook/book/blob/master/CONTRIBUTORS.md
[306] https://github.com/all-contributors
[307] https://allcontributors.org/docs/en/emoji-key

CHAPTER

FIFTYTWO

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[314]. 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:

52.1 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.

52.2 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[308]. Slides are made using reveal.js[309]. They are available as PDFs in `talks/PDFs/`, or as the source `html` files in `talks/`.

[314] 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.

[308] https://github.com/datalad-handbook/course
[309] https://github.com/hakimel/reveal.js/

52.3 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[310] tool provided in the companion course repository[311]. 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[312] 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.

52.4 Use artwork used in the handbook

The handbook's artwork[313] repository contains the sources for figures used in the handbook.

52.5 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.

[310] https://github.com/datalad-handbook/course/blob/master/tools/cast_live
[311] https://github.com/datalad-handbook/course
[312] http://handbook.datalad.org/en/latest/contributing.html#directives-and-demos
[313] https://github.com/datalad-handbook/artwork

CHAPTER

FIFTYTHREE

ACKNOWLEDGEMENTS

DataLad development is supported by a US-German collaboration in computational neuroscience (CRCNS) project “DataGit: converging catalogues, warehouses, and deployment logistics into a federated ‘data distribution’” (Halchenko[315]/Hanke[316]), co-funded by the US National Science Foundation (NSF 1429999[317]) and the German Federal Ministry of Education and Research (BMBF 01GQ1411[318]). Additional support is provided by the German federal state of Saxony-Anhalt and the European Regional Development Fund (ERDF), Project: Center for Behavioral Brain Sciences[319], Imaging Platform. This work is further facilitated by the ReproNim project (NIH 1P41EB019936-01A1[320]).

[315] http://haxbylab.dartmouth.edu/ppl/yarik.html
[316] https://www.psychoinformatics.de/
[317] https://www.nsf.gov/awardsearch/showAward?AWD_ID=1429999
[318] https://www.gesundheitsforschung-bmbf.de/de/datagit-kombination-von-katalogen-datenbanken-und-verteilungslogistik-in-eine-da
php
[319] http://cbbs.eu/en/
[320] https://projectreporter.nih.gov/project_info_description.cfm?projectnumber=1P41EB019936-01A1

germany-usa

SPONSORED BY THE

Federal Ministry of Education and Research

SACHSEN-ANHALT

Ministerium für Wissenschaft und Wirtschaft

BMBF 01GQ1411

EUROPEAN UNION

European Regional Development Fund

NSF

NSF 1429999

Part XV

Code lists from chapters

CHAPTER

FIFTYFOUR

ABOUT CODE LISTS

The following pages aggregate code examples from the book into a numbered list of easy to copy-paste code snippets. These sections are meant to be supplementary to lectures and workshops based on the handbook.

CHAPTER

FIFTYFIVE

CODE FROM CHAPTER: 01_DATASET_BASICS

Code snippet 1:

```
datalad create -c text2git DataLad-101
```

Code snippet 2:

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

Code snippet 3:

```
git log
```

Code snippet 4:

```
ls -a # show also hidden files
```

Code snippet 5:

```
mkdir books
```

Code snippet 6:

```
tree
```

Code snippet 7:

```
cd books && wget -nv https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.
↪01.pdf/download -O TLCL.pdf && wget -nv https://edisciplinas.usp.br/pluginfile.php/3252353/
↪mod_resource/content/1/b_Swaroop_Byte_of_python.pdf -O byte-of-python.pdf && cd ../
```

Code snippet 8:

```
tree
```

Code snippet 9:

```
datalad status
```

Code snippet 10:

```
datalad save -m "add books on Python and Unix to read later"
```

Code snippet 11:

```
git log -p -n 1
```

Code snippet 12:

```
cd books && wget -nv https://github.com/progit/progit2/releases/download/2.1.154/progit.pdf &
↪& cd ../
```

Code snippet 13:

```
datalad status
```

Code snippet 14:

```
datalad save -m "add reference book about git" books/progit.pdf
```

Code snippet 15:

```
# lets make the output a bit more concise with the --oneline option
git log --oneline
```

Code snippet 16:

```
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 \
```

Code snippet 17:

```
ls books
```

Code snippet 18:

```
datalad status
```

Code snippet 19:

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

EOT
```

Code snippet 20:

```
datalad status
```

Code snippet 21:

```
datalad save -m "Add notes on datalad create"
```

Code snippet 22:

```
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
```

Code snippet 23:

```
datalad status
```

Code snippet 24:

```
datalad save -m "add note on datalad save"
```

Code snippet 25:

```
git log -p -n 2
```

Code snippet 26:

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

Code snippet 27:

```
datalad clone --dataset . \
 https://github.com/datalad-datasets/longnow-podcasts.git recordings/longnow
```

Code snippet 28:

```
tree -d   # we limit the output to directories
```

Code snippet 29:

```
cd recordings/longnow/Long_Now__Seminars_About_Long_term_Thinking
ls
```

Code snippet 30:

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

Code snippet 31:

```
datalad status --annex
```

Code snippet 32:

```
datalad get Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.
↪mp3
```

Code snippet 33:

```
datalad status --annex all
```

Code snippet 34:

```
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
```

Code snippet 35:

```
git log --reverse
```

Code snippet 36:

```
# 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"
```

Code snippet 37:

```
git log -p
```

Code snippet 38:

```
cd recordings/longnow
git log --oneline
```

Code snippet 39:

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

CHAPTER

FIFTYSIX

CODE FROM CHAPTER: 02_REPRODUCIBLE_EXECUTION

Code snippet 40:

```
cd DataLad-101 && mkdir code && tree -d
```

Code snippet 41:

```
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
```

Code snippet 42:

```
datalad status
```

Code snippet 43:

```
datalad save -m "Add short script to write a list of podcast speakers and titles"
```

Code snippet 44:

```
datalad run -m "create a list of podcast titles" "bash code/list_titles.sh > recordings/
↪podcasts.tsv"
```

Code snippet 45:

```
git log -p -n 1
```

Code snippet 46:

```
datalad run -m "Try again to create a list of podcast titles" "bash code/list_titles.sh >␣
↪recordings/podcasts.tsv"
```

Code snippet 47:

```
git log --oneline
```

Code snippet 48:

```
less recordings/podcasts.tsv
```

Code snippet 49:

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

Code snippet 50:

```
datalad status
```

Code snippet 51:

```
datalad save -m "BF: list both directories content" code/list_titles.sh
```

Code snippet 52:

```
git log -n 2
```

Code snippet 53:

```
echo "$ datalad rerun $(git rev-parse HEAD~1)" && datalad rerun $(git rev-parse HEAD~1)
```

Code snippet 54:

```
git log -n 1
```

Code snippet 55:

```
datalad diff --to HEAD~1
```

Code snippet 56:

```
git diff HEAD~1
```

Code snippet 57:

```
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
```

Code snippet 58:

```
datalad save -m "add note datalad and git diff"
```

Code snippet 59:

```
git log -- recordings/podcasts.tsv
```

Code snippet 60:

```
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
```

Code snippet 61:

```
datalad save -m "add note on basic datalad run and datalad rerun"
```

Code snippet 62:

```
ls recordings/longnow/.datalad/feed_metadata/*jpg
```

Code snippet 63:

```
datalad run -m "Resize logo for slides" \
"convert -resize 400x400 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg recordings/
↪salt_logo_small.jpg"
```

Code snippet 64:

```
datalad run --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"
```

(continues on next page)

(continued from previous page)

Code snippet 65:

```
datalad run --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"
```

Code snippet 66:

```
datalad unlock recordings/salt_logo_small.jpg
```

Code snippet 67:

```
datalad status
```

Code snippet 68:

```
convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg recordings/
↪salt_logo_small.jpg
```

Code snippet 69:

```
datalad save -m "resized picture by hand"
```

Code snippet 70:

```
datalad run --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"
```

Code snippet 71:

```
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
```

Code snippet 72:

```
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"
```

Code snippet 73:

```
datalad status
```

Code snippet 74:

```
datalad save -m "add additional notes on run options"
```

Code snippet 75:

```
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"
```

Code snippet 76:

```
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
```

Code snippet 77:

```
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"
```

Code snippet 78:

```
datalad status
```

Code snippet 79:

```
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
```

Code snippet 80:

```
datalad save -m "add note on clean datasets"
```

Code snippet 81:

```
git log -p -n 2
```

CHAPTER

FIFTYSEVEN

CODE FROM CHAPTER: 10_YODA

Code snippet 123:

```
# inside of DataLad-101
datalad create -c yoda --dataset . midterm_project
```

Code snippet 124:

```
cd midterm_project
# we are in midterm_project, thus -d . points to the root of it.
datalad install -d . --source https://github.com/datalad-handbook/iris_data.git input/
```

Code snippet 125:

```
cd ../
tree -d
cd midterm_project
```

Code snippet 126:

```
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 is 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
```

(continues on next page)

(continued from previous page)

```
plot = sns.pairplot(df, hue='class')
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
```

Code snippet 127:

```
datalad status
```

Code snippet 128:

```
datalad save -m "add script for kNN classification and plotting" --version-tag␣
↪ready4analysis code/script.py
```

Code snippet 129:

```
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"
```

Code snippet 130:

```
git log --oneline
```

Code snippet 131:

```
# with the >| redirection we are replacing existing contents in the file
cat << EOT >| README.md

# Midterm YODA Data Analysis Project
```

(continues on next page)

(continued from previous page)

```
## 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
```

Code snippet 132:

```
datalad status
```

Code snippet 133:

```
datalad save -m "Provide project description" README.md
```

- genindex
- search

CHAPTER

FIFTYEIGHT

A NEUROIMAGING DATASETS

Todo: Currently, this is a left over. Later, we can rework this into something, but its unclear yet what ;-)

This section is a concise demonstration of what a DataLad dataset is, showcased on a dataset from the field of neuroimaging. A DataLad dataset is the core data type of DataLad. We will explore the concepts of it with one public example dataset, the studyforrest phase 2 data (studyforrest.org). Note that this is just one type and use of a Datalad dataset, and you throughout there are many more flavors of using DataLad datasets in the basics or in upcoming use cases.

Please follow along and run the commands below in your own terminal for a hands-on experience.

```
$ datalad install 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/me/usecases/studyforrest/studyforrest-data-
↪phase2'
[INFO]   Remote origin not usable by git-annex; setting annex-ignore
install(ok): /home/me/usecases/studyforrest/studyforrest-data-phase2 (dataset)
```

Once installed, a DataLad dataset looks like any other directory on your filesystem:

```
$ cd studyforrest-data-phase2
$ ls # output below is only an excerpt from ls
Makefile
participants.tsv
README.rst
recording-cardresp_physio.json
recording-eyegaze_physio.json
src
stimuli
sub-01
sub-02
sub-03
sub-04
```

However, all files and directories within the DataLad dataset can be tracked (should you want them to be tracked), regardless of their size. 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 datasets history.

Note for Git users:

A DataLad dataset is a Git repository. Large file content in the dataset in the annex is tracked with git-annex. An `ls -a` reveals that Git is secretly working in the background:

```
$ ls -a # show also hidden files (excerpt)
code
.datalad
dataset_description.json
.git
.gitattributes
.gitignore
.gitmodules
participants.tsv
README.rst
recording-cardresp_physio.json
recording-eyegaze_physio.json
src
stimuli
sub-01
sub-02
sub-03
sub-04
sub-05
```

Users can *create* new DataLad datasets from scratch, or install existing DataLad datasets from paths, urls, or open-data collections. This makes sharing and accessing data fast and easy. Moreover, when sharing or installing a DataLad dataset, all copies also include the datasets history. An installed DataLad dataset knows the dataset it was installed from, and if changes in this original DataLad dataset happen, the installed dataset can simply be updated.

You can view the DataLad datasets history with tools of your choice. The code block below is used to illustrate the history and is an exempt from **`git log`**.

```
$ git log --oneline --graph --decorate
* a6623bff (HEAD -> master, origin/master, origin/HEAD) [DATALAD] dataset aggregate metadata␣
↪update
* 72d535d5 Enable DataLad metadata extractors
* 9c15094e [DATALAD] new dataset
* d97455f5 [DATALAD] Set default backend for all files to be MD5E
* e2a2cf1c Update changelog for 1.5
* 6da25fb6 BF: Re-import respiratory trace after bug fix in converter (fixes gh-11)
* 131edcb7 Fix type in physio log converter (fixes gh-11)
* fbb5d619 ENH: Report per-stimulus events (fixes gh-6)
* d6f3fcd2 Add BIDS-compatible stimuli/ directory (with symlinks)
```

58.1 Dataset content identity and availability information

Upon installation of a DataLad dataset, DataLad retrieves only (small) metadata information about the dataset. This exposes the datasets file hierarchy for exploration, and speeds up the installation of a DataLad dataset of many TB in size to a few seconds. Just after installation, the dataset is small in size:

```
$ du -sh
17M       .
```

This is because only small files are present locally – for shits and giggles, you can try opening both a small `.tsv` file in the root of the dataset, and a larger compressed `nifti` (`nii.gz`) in one of the subdirectories in this dataset. A small `.tsv` (1.9K) file exists and can be opened locally, but what would be a large, compressed `nifti` file is not. In this state, one cannot open or work with the nifti file, but you can explore which files exist without the potentially large download.

```
$ ls participants.tsv  sub-01/ses-movie/func/sub-01_ses-movie_task-movie_run-1_bold.nii.gz
participants.tsv
sub-01/ses-movie/func/sub-01_ses-movie_task-movie_run-1_bold.nii.gz
```

The retrieval of the actual, potentially large file content can happen at any later time for the full dataset or subsets of files. Let's get the nifti file:

```
$ datalad get sub-01/ses-movie/func/sub-01_ses-movie_task-movie_run-1_bold.nii.gz
get(ok): sub-01/ses-movie/func/sub-01_ses-movie_task-movie_run-1_bold.nii.gz (file) [from␣
↪mddatasrc...]
```

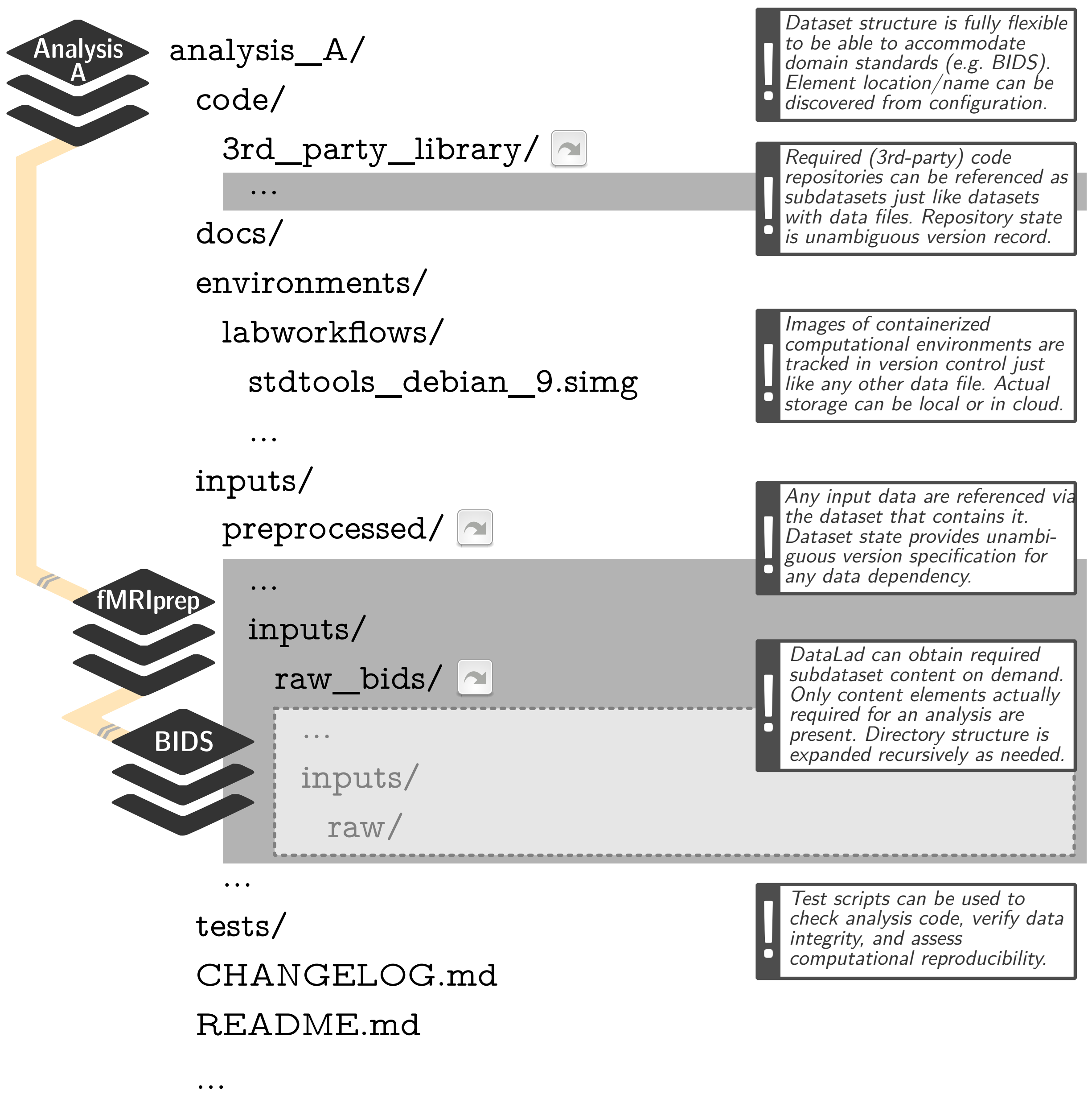
Wasn't this easy?

58.2 Dataset Nesting

Within DataLad datasets one can *nest* other DataLad datasets arbitralily deep. This does not seem particulary spectacular - after all, any directory on a filesystem can have other directories inside it. The possibility for nested Datasets, however, is one of many advantages DataLad datasets have: 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.

By taking advantage of dataset nesting, one can take datasets such as the studyforrest phase-2 data and install it as a subdataset within a superdataset containing analysis code and results computed from the studyforrest data. Should the studyforrest data get extended or changed, its subdataset can be updated to include the changes easily. More detailed examples of this can be found in the use cases in the last section (for example in *Writing a reproducible paper* (page 314)).

The figure below illustrates dataset nesting in a neuroimaging context schematically:



Dataset structure is fully flexible to be able to accommodate domain standards (e.g. BIDS). Element location/name can be discovered from configuration.

Required (3rd-party) code repositories can be referenced as subdatasets just like datasets with data files. Repository state is unambiguous version record.

Images of containerized computational environments are tracked in version control just like any other data file. Actual storage can be local or in cloud.

Any input data are referenced via the dataset that contains it. Dataset state provides unambiguous version specification for any data dependency.

DataLad can obtain required subdataset content on demand. Only content elements actually required for an analysis are present. Directory structure is expanded recursively as needed.

Test scripts can be used to check analysis code, verify data integrity, and assess computational reproducibility.

58.3 Creating your own dataset yourself

Anyone can create, populate, and optionally share a *new* DataLad dataset. A new DataLad dataset is always created empty, even if the target directory already contains additional files or directories. After creation, arbitralily large amounts of data can be added. Once files are added and saved to the dataset, any changes done to these data files can be saved to the history.

Note for Git users:

Creation of datasets relies on the **`git init`** and **`git annex init`** commands.

As already shown, already existing datalad dataset can be simply installed from a url or path, or from the datalad open-data collection.

Note for Git users:

`datalad install` used the **`git clone`** command.

INDEX