1. 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 (www.datalad.org) is a software tool developed to aid with everything related to the evolution of digital objects.

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

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

1.1. On data

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

Most data need to be shared – may it be a digital collection of family photos, a genomics 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 copies of one state of their history, keeping all shared copies up-to-date once the original data change or evolve 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: 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 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.

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, organize your private music library, keep track of all cat memes on the internet, or anything else.

A dataset is a Git repository. All features of the version control system Git also apply to everything managed by DataLad – plus many more. If you do not know or use Git yet, there is no need to panic – there is no necessity to learn all of Git to follow along in learning and using DataLad. You will experience much of Git working its magic underneath the hood when you use DataLad, and will soon start to appreciate its

features. Later, you may want to know more on how DataLad uses Git as a fundamental layer and learn some of Git.

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!

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.

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 web server, or any other common computational infrastructure you might be using.

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, Figshare

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 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.2. How to use the handbook

2.1. For whom this book is written

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

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

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

2.2. How to read this book

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

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

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.
# Otherwise, anything preceded by $ is a command to try.
# If the line starts with no $, it is an output of a command.
datalad create myfirstrepo
[INFO   ] Creating a new annex repo at /home/me/DataLad-101
create(ok): /home/me/DataLad-101 (dataset)
```

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

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

Whenever you see a ✂ symbol, command output has been shortened for better readability. In the example below, the commit shasum has been shortened and marked with ✂SHA1.

```
git log --reverse

commit 8df130bb✂SHA1

Author: Elena Piscopia <elena@example.net>

Date:   Tue Jun 18 16:13:00 2019 +0000
```

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

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

“Find-out-more” boxes contain general additional information:

Click here to show/hide further commands

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

For (future) Git experts

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

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

For Windows users only

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, write or request an additional chapter if you believe it is a worthwhile addition, or ask a question on Neurostars.org with a datalad tag if you need help.

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

../_images/student.svg

I cannot/do not want to code along…

2.3. Let’s get going!

If you have DataLad installed, you can dive straight into chapter Create a dataset. For everyone new, there are the sections The command line as a minimal tutorial to using the shell and Installation and configuration to get your DataLad installation set up.3. Installation and configuration

3.1. Install DataLad

Feedback on installation instructions

The installation methods presented in this chapter are based on experience and have been tested carefully. However, operating systems and other software are continuously evolving, and these guides might have become outdated. Be sure to check out the online-handbook for up-to-date information.

In general, the DataLad installation requires Python 3 (see the Find-out-more on the difference between Python 2 and 3 to learn why this is required), Git, and git-annex, and for some functionality 7-Zip. The instructions below detail how to install the core DataLad tool and its dependencies on common operating systems. They do not cover the various DataLad extensions that need to be installed separately, if desired.

Python 2, Python 3, what's the difference?

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

Cartoon of a person sitting on the floor in front of a laptop

3.1.1. Windows 10 and 11

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

Python:

Windows itself does not ship with Python, it must be installed separately. If you already did that, please check the Find-out-more on Python versions, if it matches the requirements. Otherwise, head over to the download section of the Python website, and download an installer. Unless you have specific requirements, go with the 64bit installer of the latest Python 3 release.

Avoid installing Python from the Windows store

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

Verify Python installation

Git:

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

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

Select Git from the command line and also from 3rd-party software

Enable file system caching

Select Use external OpenSSH

Enable symbolic links

Git-annex:

There are two convenient ways to install git-annex. The first is downloading the installer from git-annex' homepage. The other is to deploy git-annex via the DataLad installer. The latter option requires the installation of the datalad-installer Python package. Once Python is available, it can be done with the Python package manager pip. Open a command prompt and run:

```
> python -m pip install datalad-installer
```

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

```
> datalad-installer git-annex -m datalad/git-annex:release
```

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

For performance improvements, regardless of which installation method you chose, we recommend to also set the following git-annex configuration:

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

DataLad:

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

```
> python -m pip install datalad
```

7-Zip (optional, but highly recommended):

Download it from the 7-zip website (64bit installer when in doubt), and install it into the default target directory.

There are many other ways to install DataLad on Windows, check for example the Windows-wit on the Windows Subsystem 2 for Linux. One attractive alternative approach is Conda, a completely different approach is to install the DataLad Gooey, which is a standalone installation of DataLad's graphical application (see the DataLad Gooey documentation for installation instructions).

Install DataLad using the Windows Subsystem 2 for Linux

Using DataLad on Windows has a few peculiarities. In general, DataLad can feel a bit sluggish on non-WSL2 Windows systems. This is due to various file system issues that also affect the version control system Git itself, which DataLad relies on. The core functionality of DataLad works, and you should be able to follow most contents covered in this book. You will notice, however, that some Unix commands displayed in examples may not work, and that terminal output can look different from what is displayed in the code examples of the book, and that some dependencies for additional functionality are not available for Windows. Dedicated notes, "Windows-wits", contain important information, alternative commands, or warnings, and an overview of useful Windows commands and general information is included in The command line.

3.1.2. Mac (incl. M1)

Modern Macs come with a compatible Python 3 version installed by default. The Find-out-more on Python versions has instructions on how to confirm that.

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

Next, install datalad and its dependencies:

brew install datalad

Alternatively, you can exclusively use brew for DataLad's non-Python dependencies, and then check the Find-out-more on how to install DataLad via Python's package manager.

Install DataLad via pip on macOS

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

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

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

Whichever repository you end up using, the following command installs DataLad and all of its software dependencies (including git-annex and p7zip):

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

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

3.1.4. Linux: CentOS, Redhat, Fedora, or similar systems

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

DataLad itself can be installed via pip:

```
python -m pip install datalad
```

Alternatively, DataLad can be installed together with Git and git-annex via Conda.

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

The most convenient user-based installation can be achieved via Conda.

3.1.6. Conda

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

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

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

bash Miniconda3-latest-<YOUR-OS>-x86_64.sh

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

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

conda install -c conda-forge datalad

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

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh

bash Miniconda3-latest-Linux-x86_64.sh

acknowledge license, keep everything at default

conda install -c conda-forge datalad

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

To update an existing installation with conda, use:

```
conda update -c conda-forge datalad
```

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

3.1.7. Using Python's package manager pip

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

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

```
python -m pip install datalad
```

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

```
python -m pip install --user datalad
```

On some systems, you may need to call python3 instead of python:

```
python3 -m pip install datalad
# or, in case of a "permission denied error":
python3 -m pip install --user datalad
```

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

pip is not able to install non-Python software, such as 7-zip or git-annex. But you can install the DataLad installer via a python -m pip install datalad-installer. This is a

command-line tool that aids installation of DataLad and its key software dependencies on a range of platforms.

3.2. 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.

../_images/gitidentity.svg

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.5. What you really need to know

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

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

5.1. DataLad datasets

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

Create DataLad datasets

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

5.2. Simplified local version control workflows

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

Version control arbitrarily large contents

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.

Consume and collaborate

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

5.4. Dataset linkage

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

Dataset nesting

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).

provenance capture

You or your collaborators can thus reobtain or reproducibly recompute content with a single command, and make use of extensive provenance of dataset content (who created it, when, and how?).

5.6. Third party service integration

Export datasets to third party services such as GitHub, GitLab, or Figshare with built-in commands.

third party integration

Alternatively, you can use a multitude of other available third party services such as Dropbox, Google Drive, Amazon S3, owncloud, 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.

meta data capabilities

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, and a concise command cheat-sheet in section DataLad cheat sheet.

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

Basics 1. DataLad datasets Previous: 1.1. Create a dataset Next: 1.3. Modify content Quick search 1.2. Populate a dataset The first lecture in DataLad-101 referenced some useful literature. Even if we end up not reading those books at all, let’s download them nevertheless and put them into our dataset. You never know, right? Let’s first create a

directory to save books for additional reading in. mkdir books Let's take a look at the current directory structure with the tree command[1]: 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 An intro to Python: A byte of Python You can either visit the links and save them in books/, or run the following commands[2] to download the books right from the terminal. Note that we line break the command with \ line continuation characters. In your own work you can write commands like this into a single line. If you copy them into your terminal as they are presented here, make sure to check the Windows-wit on peculiarities of its terminals. Terminals other than Git Bash can't handle multi-line commands cd books wget -q https://sourceforge.net/projects/linuxcommand/files/TLCL/19.01/TLCL-19.01.pdf/download \ -O TLCL.pdf wget -q https://homepages.uc.edu/~becktl/byte_of_python.pdf \ -O byte-of-python.pdf # get back into the root of the dataset cd ../ Some machines will not have wget available by default, but any command that can download a file can work as an alternative. See the Windows-wit for the popular alternative curl. You can use curl instead of wget 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 (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 (manual) command. This time, it is your turn to specify a helpful commit message with the -m option (although the DataLad command is datalad save, we talk about commit messages because datalad save ultimately uses the command git commit (manual) to do its work): datalad save -m "add books on Python and Unix to read later" add(ok): books/TLCL.pdf (file) add(ok): books/byte-of-python.pdf (file) save(ok): . (dataset) If you ever forget to specify a message, or made a typo, not all is lost. A Find-out-more explains how to amend a saved state. "Oh no! I forgot the -m option for 'datalad save'!"

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 (manual). 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 b40316a6✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 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--06d1efcb✂MD5.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..7a6e51e --- /dev/null +++ b/books/byte-of-python.pdf Now this might look a bit cryptic (and honestly, tig[3] makes it look prettier). But this tells us the date and time in which a particular author added two PDFs to the directory books/, and thanks to that commit message we have a nice human-readable summary of that action. A Find-out-more explains what makes a good message. DOs and DON'Ts for commit messages There is no staging area in DataLad Just as in Git, new files are not tracked from their creation on, but only when explicitly added to Git (in Git terms, with an initial git add (manual)). 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 are absolutely right! It is good practice to save only those changes together that belong together. We do not want to squish completely unrelated changes into the same spot of our history, because it would get very nasty

should we want to revert some of the changes without affecting others in this commit. Luckily, we can point datalad save to exactly the changes we want it to record. Let's try this by adding yet another book, a good reference work about git, Pro Git: cd books wget -q https://github.com/progit/progit2/releases/download/2.1.154/progit.pdf cd ../ datalad status shows that there is a new untracked file: datalad status untracked: books/progit.pdf (file) Let's give datalad save precisely this file by specifying its path after the commit message: datalad save -m "add reference book about git" books/progit.pdf add(ok): books/progit.pdf (file) save(ok): . (dataset) Regarding your second remark, you are right that a datalad save without a path specification would write all of the currently untracked files or modifications to the history. But check the Find-out-more on how to tell it otherwise. How to save already tracked dataset components only? 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 a875e49 add reference book about git b40316a add books on Python and Unix to read later e0ff3a7 Instruct annex to add text files to Git 4ce681d [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). 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 (manual) command. We will dive deeper into the provenance-related benefits of using it in later chapters, but for now, we'll start with best-practice-building. datalad download-url can retrieve content from a URL (following any URL-scheme from https, http, or ftp or s3) and save it into the dataset together with a human-readable commit message and a hidden, machine-readable record of the origin of the content. This saves you time, and captures provenance information about the data you add to your dataset. To experience this, let's add a final book, a beginner's guide to bash, to the dataset. We provide the command with a URL, a pointer to the dataset the file should be saved in (. denotes "current directory"), and a commit message. datalad download-url \

https://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf \ --dataset . \ -m "add beginners guide on bash" \ -O 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) 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 nothing to save, working tree clean This is because datalad download-url took care of saving for you: git log -p -n 1 commit 59ac8d32✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 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--0ab2c121✂MD5.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 59ac8d3 add beginners guide on bash a875e49 add reference book about git b40316a add books on Python and Unix to read later e0ff3a7 Instruct annex to add text files to Git 4ce681d [DATALAD] new dataset Well done! Your DataLad-101 dataset and its history are slowly growing. Footnotes [1] tree is a Unix command to list file system content. If it is not yet installed, you can get it with your native package manager (e.g., apt, brew, or conda). For example, if you use OSX, brew install tree will get you this tool. Windows has its own tree command. Note that this tree works slightly different than its Unix equivalent - by default, it will only display directories, not files, and the command options it accepts are either /f (display file names) or /a (change display of subdirectories to text instead of graphic characters). [2] 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. [3] See tig. Once installed, exchange any git log command you see here with the single word tig. ← 1.1. Create a dataset 1.3. Modify content → v: latest The DataLad Handbook Related Topics Documentation overview Basics 1. DataLad datasets Previous: 1.2. Populate a dataset Next: 1.4. Install datasets Quick search 1.3. Modify content So far, we've only added new content to the dataset. And we have not done much to that content up to this point, to be honest. Let's see what happens if we add content, and then modify it. For this, in the root of DataLad-101,

create a plain text file called notes.txt. It will contain all of the notes that you take throughout the course. Let's write a short summary of how to create a DataLad dataset from scratch: "One can create a new dataset with 'datalad create [–description] PATH'. The dataset is created empty". This is meant to be a note you would take in an educational course. You can take this note and write it to a file with an editor of your choice. The code snippet, however, contains this note within the start and end part of a heredoc. 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. How does a heredoc (here-document) work? Running this command will create notes.txt in the root of your DataLad-101 dataset: Heredocs don't work under non-Git-Bash Windows terminals cat << EOT > notes.txt One can create a new dataset with 'datalad create [--description] PATH'. The dataset is created empty EOT Run datalad status (manual) 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) 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 (manual), so write a short summary on that as well. Again, the example uses Unix commands (cat and redirection, this time however with >> to append new content to the existing file) to accomplish this, but you can take any editor of your choice. cat << EOT >> notes.txt The command "datalad save [-m] PATH" saves the file (modifications) to history. Note to self: Always use informative, concise commit messages. EOT Let's check the dataset's current state: datalad status modified: notes.txt (file) and save the file in DataLad: datalad save -m "add note on datalad save" add(ok): notes.txt (file) save(ok): . (dataset) Let's take another look into our history to see the development of this file. We are using git log -p -n 2 (manual) 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 e310b465✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 add note on datalad save diff --git a/notes.txt b/notes.txt index 3a7a1fe..0142412 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 874d766f✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 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? 'git log' has many more useful options ← 1.2. Populate a dataset 1.4. Install datasets → v: latest The DataLad Handbook Table of Contents 1.4. Install datasets 1.4.1. Dataset content identity and availability information 1.4.2. Keep whatever you like 1.4.3. Dataset archeology Related Topics Documentation overview Basics 1. DataLad datasets Previous: 1.3. Modify content Next: 1.5. Dataset nesting Quick search 1.4. Install datasets So far, we have created a DataLad-101 course dataset. We saved some additional readings into the dataset, and have carefully made and saved notes on the DataLad commands we discovered. Up to this point, we therefore know the typical, local workflow to create and populate a dataset from scratch. But we've been told that with DataLad we could very easily get vast amounts of data to our computer. Rumor has it that this would be only a single command in the terminal! Therefore, everyone in today's lecture excitedly awaits today's topic: Installing datasets. "With DataLad, users can install clones of existing DataLad datasets from paths, URLs, or open-data collections" our lecturer begins. "This makes accessing data fast and easy. A dataset that others could install can be created by anyone, without a need for additional software. Your own datasets can be installed by others, should you want that, for example. Therefore, not only accessing data becomes fast and easy, but also sharing." "That's so cool!", you think. "Exam preparation will be a piece of cake if all of us can share our mid-term and final projects easily!" "But today, let's only focus on how to install a dataset", she continues. "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 cannot 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 are 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[2]. It consists of audio recordings about long-term thinking, and while the

DataLad-101 course is not a long-term thinking seminar, those recordings are nevertheless a good addition to the large stash of yet-to-read text books we piled up. Let's get this dataset into our existing DataLad-101 dataset. To keep the DataLad-101 dataset neat and organized, we first create a new directory, called recordings. # we are in the root of DataLad-101 mkdir recordings The command that can be used to obtain a dataset is datalad clone (manual), but we often refer to the process of cloning a Dataset as installing. Let's install the longnow podcasts in this new directory. The datalad clone 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[1]. 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 as second positional argument a path to the destination, – a path to where we want to install the dataset to. In this case it is recordings/longnow. Because we are installing a dataset (the podcasts) into an existing dataset (the DataLad-101 dataset), we also supply a -d/--dataset flag to the command. This specifies the dataset to perform the operation on, and allows us to install the podcasts as a subdataset of DataLad-101. Because we are in the root of the DataLad-101 dataset, the pointer to the dataset is a . (which is Unix' way of saying "current directory"). As before with long commands, we line break the code 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] Remote origin not usable by git-annex; setting annex-ignore install(ok): recordings/longnow (dataset) add(ok): recordings/longnow (dataset) add(ok): .gitmodules (file) save(ok): . (dataset) add(ok): .gitmodules (file) save(ok): . (dataset) This command copied the repository found at the URL https://github.com/datalad-datasets/longnow-podcasts into the existing DataLad-101 dataset, into the directory recordings/longnow. The optional destination is helpful: If we had not specified the path recordings/longnow as a destination for the dataset clone, the command would have installed the dataset into the root of the DataLad-101 dataset, and instead of longnow it would have used the name of the remote repository "longnow-podcasts". But the coolest feature of datalad clone is yet invisible: This command also recorded where this dataset came from, thus capturing its origin as provenance. Even though this is not obvious at this point in time, later chapters in this handbook will demonstrate how useful this information can be. Clone internals The datalad clone command uses git clone (manual). A dataset that is installed from an

existing source, e.g., a path or URL, is the DataLad equivalent of a clone in Git. Do I have to install from the root of datasets? What if I do not install into an existing dataset? Here is the repository structure: use tree 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 with two subdirectories. If we navigate into one of them and list its content, we'll see many .mp3 files (here is an excerpt). cd recordings/longnow/Long_Now__Seminars_About_Long_term_Thinking ls 2003_11_15__Brian_Eno__The_Long_Now.mp3 2003_12_13__Peter_Schwartz__The_Art_Of_The_Really_Long_View.mp3 2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3 2004_02_14__James_Dewar__Long_term_Policy_Analysis.mp3 2004_03_13__Rusty_Schweickart__The_Asteroid_Threat_Over_the_Next_100_000_Years.mp3 2004_04_10__Daniel_Janzen__Third_World_Conservation__It_s_ALL_Gardening.mp3 2004_05_15__David_Rumsey__Mapping_Time.mp3 2004_06_12__Bruce_Sterling__The_Singularity__Your_Future_as_a_Black_Hole.mp3 2004_07_10__Jill_Tarter__The_Search_for_Extra_terrestrial_Intelligence__Necessarily_a_Long_term_Strategy.mp3 2004_08_14__Phillip_Longman__The_Depopulation_Problem.mp3 2004_09_11__Danny_Hillis__Progress_on_the_10_000_year_Clock.mp3 2004_10_16__Paul_Hawken__The_Long_Green.mp3 2004_11_13__Michael_West__The_Prospects_of_Human_Life_Extension.mp3 1.4.1. 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 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 (manual). Make sure to be (somewhere) inside of the longnow dataset to execute the following command: datalad status --annex 236 annex'd files (15.4 GB recorded total size) nothing to save, working tree clean Woah! More than 200 files, totaling more than 15 GB? You begin to appreciate that DataLad did not download all of this data right away! That would have taken hours given the crappy internet connection in the lecture hall, and you are not even sure whether your hard drive has much space left… But you nevertheless are curious on how to actually listen to one of these .mp3s now. So how does one actually "get" the files? The command to retrieve file content is datalad get (manual). You can specify one or more specific files, or get all of the dataset by specifying datalad get . at the root directory of the dataset (with . denoting "current directory"). First, we get one of the recordings in the dataset – take any one of your choice (here, it's the first). datalad get Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.mp3 get(ok): Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.mp3 (file) [from web...] Try to open it – it will now work. If you would want to get the rest of the missing data, instead of specifying all files individually, we can use . to refer to all of the dataset like this: datalad get . However, with a total size of more than 15GB, this might take a while, so do not do that now. If you did execute the command above, interrupt it by pressing CTRL + C – Do not worry, this will not break anything. Isn't that easy? Let's see how much content is now present locally. For this, datalad status --annex all has a nice summary: datalad status --annex all 236 annex'd files (35.7 MB/15.4 GB present/total size) nothing to save, working tree clean This shows you how much of the total content is present locally. With one file, it is only a fraction of the total size. Let's get a few more recordings, just because it was so mesmerizing to watch DataLad's fancy progress bars. datalad get Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_N

ow.mp3 \
Long_Now__Seminars_About_Long_term_Thinking/2003_12_13__Peter_Schwartz__The_Art_Of_The_Really_Long_View.mp3 \
Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3
get(ok):
Long_Now__Seminars_About_Long_term_Thinking/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. Get internals datalad get uses git annex get (manual) underneath the hood. 1.4.2. Keep whatever you like “Oh shit, oh shit, oh shit…” you hear from right behind you. Your fellow student apparently downloaded the full dataset accidentally. “Is there a way to get rid of file contents in dataset, too?”, they ask. “Yes”, the lecturer responds, “you can remove file contents by using datalad drop (manual). This is really helpful to save disk space for data you can easily reobtain, for example”. The datalad drop command will remove file contents completely from your dataset. You should only use this command to remove contents that you can datalad get again, or generate again (for example, with next chapter’s datalad datalad run (manual) command), or that you really do not need anymore. Let’s remove the content of one of the files that we have downloaded, and check what this does to the total size of the dataset. Here is the current amount of retrieved data in this dataset: datalad status --annex all 236 annex'd files (135.1 MB/15.4 GB present/total size) nothing to save, working tree clean We drop a single recording’s content that we previously downloaded with datalad get … datalad drop Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3
drop(ok):
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) … and check the size of the dataset again: datalad status --annex all 236 annex'd files (93.5 MB/15.4 GB present/total size) nothing to save, working tree clean Dropping

the file content of one mp3 file saved roughly 40MB of disk space. Whenever you need the recording again, it is easy to re-retrieve it: datalad get Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3 get(ok): Long_Now__Seminars_About_Long_term_Thinking/2004_01_10__George_Dyson__There_s_Plenty_of_Room_at_the_Top__Long_term_Thinking_About_Large_scale_Computing.mp3 (file) [from web...] Reobtained! This was only a quick digression into datalad drop. The main principles of this command will become clear after chapter Under the hood: git-annex, and its precise use is shown in the paragraph on removing file contents. At this point, however, you already know that datasets allow you do datalad drop file contents flexibly. If you want to, you could have more podcasts (or other data) on your computer than you have disk space available by using DataLad datasets – and that really is a cool feature to have. 1.4.3. Dataset archeology You have now experienced how easy it is to (re)obtain shared data with DataLad. But beyond sharing only the data in the dataset, when sharing or installing a DataLad dataset, all copies also include the dataset's 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 8df130bb✂SHA1 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 3d0dc8f5✂SHA1 Author: Michael Hanke <michael.hanke@gmail.com> Date: Mon Jul 16 16:08:24 2018 +0200 [DATALAD] new dataset But that's not all. The seminar series is ongoing, and more recordings can get added to the original repository shared on GitHub. Because an installed dataset knows the dataset it was installed from, your local dataset clone can be updated from its origin, and thus get the new recordings, should there be some. Later in this handbook, we will see examples of this. Now you can not only create datasets and work with them locally, you can also consume existing datasets by installing them. Because that's cool, and because you will use this command frequently, make a note of it into your notes.txt, and datalad save (manual) the modification. # in the root of DataLad-101: cd ../../ cat << EOT >> notes.txt The command 'datalad clone URL/PATH [PATH]' installs a dataset from e.g., a URL or a path. If you install a dataset into an existing dataset (as a subdataset), remember to specify the root of the superdataset with the '-d' option. EOT datalad save -m "Add note on datalad clone" add(ok): notes.txt (file) save(ok): . (dataset) Empty files can be confusing Listing files directly after the installation of a dataset will work if done

in a terminal with ls. However, certain file managers (such as OSX's Finder[3]) 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. Consider browsing datasets with the DataLad Gooey to be on the safe side. More about why this is will be explained in section Data integrity. Footnotes [1] 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. [2] The longnow podcasts are lectures and conversations on long-term thinking produced by the LongNow foundation and we can wholeheartedly recommend them for their worldly wisdoms and compelling, thoughtful ideas. Subscribe to the podcasts at https://longnow.org/seminars/podcast. Support the foundation by becoming a member: https://longnow.org/join. [3] You can also upgrade your file manager to display file types in a DataLad datasets (e.g., with the git-annex-turtle extension for Finder) ← 1.3. Modify content 1.5. Dataset nesting → v: latest The DataLad Handbook Related Topics Documentation overview Basics 1. DataLad datasets Previous: 1.4. Install datasets Next: 1.6. Summary Quick search 1.5. Dataset nesting Without noticing, the previous section demonstrated another core principle and feature of DataLad datasets: Nesting. Within DataLad datasets one can nest other DataLad datasets arbitrarily deep. We for example just installed one dataset, the longnow podcasts, into another dataset, the DataLad-101 dataset. This was done by supplying the --dataset/-d flag in the command call. At first glance, nesting does not seem particularly spectacular – after all, any directory on a file system can have other directories inside of it. The possibility for nested Datasets, however, is one of many advantages DataLad datasets have: One aspect of nested datasets is that any DataLad dataset (subdataset or superdataset) keeps their stand-alone history. The top-level DataLad dataset (the superdataset) only stores which version of the subdataset is currently used through an identifier. 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 reuse of any other DataLad dataset, and this reuse is possible and simple precisely because all of the information is kept within a (sub)dataset. But now let's also check out how the superdataset's (DataLad-101) history

looks like after the addition of a subdataset. To do this, make sure you are outside of the subdataset longnow. Note that the first commit is our recent addition to notes.txt, so we'll look at the second most recent commit in this excerpt. git log -p -n 3 commit 3c016f73✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [DATALAD] Added subdataset diff --git a/.gitmodules b/.gitmodules new file mode 100644 index 0000000..9bc9ee9 --- /dev/null +++ b/.gitmodules @@ -0,0 +1,5 @@ +[submodule "recordings/longnow"] + path = recordings/longnow + url = https://github.com/datalad-datasets/longnow-podcasts.git + datalad-id = b3ca2718-8901-11e8-99aa-a0369f7c647e + datalad-url = https://github.com/datalad-datasets/longnow-podcasts.git 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 dcc34fbe✂SHA1 commit e310b465✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 add note on datalad save diff --git a/notes.txt b/notes.txt index 3a7a1fe..0142412 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. We have highlighted the important part of this rather long commit summary. Note that you cannot see any .mp3s being added to the dataset, as was previously the case when we datalad save (manual)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 the identifier that DataLad internally used to identify the files and the changes to the files in the subdataset. Exactly this shasum is what identifies the state of the subdataset. Navigate back into longnow and try to find the highlighted shasum in the subdataset's history: cd recordings/longnow git log --oneline dcc34fb Update aggregated metadata 36a30a1 [DATALAD RUNCMD] Update from feed bafdc04 Uniformize JSON-LD context with DataLad's internal extractors 004e484 [DATALAD RUNCMD] .datalad/maint/make_readme.py 7ee3ded Sort episodes newest-first e829615 Link to the handbook as a source of wisdom 4b37790 Fix README generator to parse correct directory We can see that it is the most recent commit shasum of the subdataset (albeit we can see only the first seven characters here – a git log (manual) would show you the full shasum). Thus, your dataset does not only know the origin of its subdataset, but also which version of the subdataset to use, i.e., it has the identifier of the stage/version in the subdataset's evolution to be used. This is what is meant by "the top-

level DataLad dataset (the superdataset) only stores which version of the subdataset is currently used through an identifier". Importantly, once we learn how to make use of the history of a dataset, we can set subdatasets to previous states, or update them. Do I have to navigate into the subdataset to see it's history? 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, Fig. 1.1 illustrates the current state of our dataset, DataLad-101, with its nested subdataset. 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. ../_images/virtual_dstree_dl101.svg Fig. 1.1 Virtual directory tree of a nested DataLad dataset If you have executed the above code snippets, make sure to go back into the root of the dataset again: cd ../../ ← 1.4. Install datasets 1.6. Summary → v: latest The DataLad Handbook Table of Contents 1.6. Summary 1.6.1. Now what can I do with that? Related Topics Documentation overview Basics 1. DataLad datasets Previous: 1.5. Dataset nesting Next: 2. DataLad, run! Quick search 1.6. Summary In the last few sections, we have discovered the basics of starting a DataLad dataset from scratch, and making simple modifications locally. An empty dataset can be created with the datalad create (manual) 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 (manual)) or external tools (such as tig) allow to explore the history. The datalad save (manual) 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: A simple, local version control workflow with datalad. Fig. 1.2 A simple, local version control workflow with DataLad. datalad status (manual) 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 (manual) can retrieve files from websources and save them automatically to your dataset. This does not only save you the time of one datalad save,

but it also records the source of the file as hidden provenance information. Furthermore, we have discovered the basics of installing a published DataLad dataset, and experienced the concept of modular nesting datasets. A published dataset can be installed with the datalad clone (manual) 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 another dataset as a subdataset, the --dataset/-d option needs to specify the root of the containing dataset, the superdataset. The source can be a URL, for example of a GitHub repository as in section Install datasets, 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 (manual) 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 stores the currently used version of a contained subdataset through an identifier. 1.6.1. Now what can I do with that? Simple, local workflows allow you to version control changing small files, for example, your CV, your code, or a book that you are working on, but you can also add very large files to your datasets history. Currently, this can be considered "best-practice building": Frequent datalad status commands, datalad save commands to save dataset modifications, and concise commit messages are the main take always 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 reuse datasets in a modular fashion. While this may appear abstract, upcoming sections will demonstrate many examples of why this can be

handy. ← 1.5. Dataset nesting 2. DataLad, run! → v: latest Related Topics Documentation overview Basics 2. DataLad, run! Previous: 2. DataLad, run! Next: 2.2. DataLad, rerun! Quick search 2.1. Keeping track In previous examples, with the exception of datalad download-url (manual), 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. 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 are 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 are following a best practice that will reappear in the later section on YODA principles: Collecting all additional scripts that work with content of a subdataset outside of this subdataset, in a dedicated code/ directory, and collating the output of the execution of these scripts outside of the subdataset as well – and therefore not modifying the subdataset. The motivation behind this will become clear in later sections, but for now we'll start with best-practice building. Therefore, create a subdirectory code/ in the DataLad-101 superdataset: $ mkdir code $ tree -d . ├── books ├── code └── recordings └── longnow ├── Long_Now__Conversations_at_The_Interval └── Long_Now__Seminars_About_Long_term_Thinking 6 directories Inside of DataLad-101/code, create a simple shell script list_titles.sh. This script will carry out a simple task: It will loop through the file names of the .mp3 files and write out speaker names and talk

titles in a very basic fashion. The cat command will write the script content into code/list_titles.sh. Here's a script for Windows users Be mindful of hidden extensions when creating files! $ 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) 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 (manual) command 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 datalad save (manual) 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) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [bash code/list_titles.sh > recordings/po...] add(ok): recordings/podcasts.tsv (file) save(ok): . (dataset) Let's take a look into the history: $ git log -p -n 1 # On Windows, you may just want to type "git log". commit e37c9fc9✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [DATALAD RUNCMD] create a list of podcast titles === Do not change lines below === { "chain": [], "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv", "dsid": "e3e70682-c209-4cac-629f-6fbed82c07cd", "exit": 0, "extra_inputs": [], "inputs": [],

"outputs": [], "pwd": "." } ^^^ Do not change lines above ^^^ diff --git a/recordings/podcasts.tsv b/recordings/podcasts.tsv new file mode 100644 index 0000000..f691b53 --- /dev/null +++ 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 (manual) 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 that 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) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [bash code/list_titles.sh > recordings/po...] $ git log --oneline e37c9fc [DATALAD RUNCMD] create a list of podcast titles e799b6b Add short script to write a list of podcast speakers and titles 87609a3 Add note on datalad clone 3c016f7 [DATALAD] Added subdataset 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. ← 2. DataLad, run! 2.2. DataLad, rerun! → v: latest The DataLad Handbook Related Topics Documentation overview Basics 2. DataLad, run! Previous: 2.1. Keeping track Next: 2.3. Input and output Quick search 2.2. DataLad, rerun! 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 2004-02-14 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 -✂--✂- 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 (manual) 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: Here's a script adjustment for Windows users 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) 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 (manual) command! That's it!", he says happily. So you go ahead and find the commit shasum in your history: git log -n 2 commit f7ea9f3d✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 BF: list both directories content commit e37c9fc9✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [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 are using all of them). datalad rerun e37c9fc9✂SHA1 [INFO] run commit e37c9fc; (create a list of ...) [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [bash code/list_titles.sh > recordings/po...] add(ok): recordings/podcasts.tsv (file) save(ok): . (dataset) action summary: add (ok: 1) run (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 08120c38✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [DATALAD RUNCMD] create a list of podcast titles === Do not change lines below === { "chain": ["e37c9fc9✂SHA1"], "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv", "dsid": "e3e70682-c209-4cac-629f-6fbed82c07cd", "exit": 0, "extra_inputs": [], "inputs": [], "outputs": [], "pwd": "." } ^^^ Do not change lines above ^^^ In the dataset's history, we can see that a new datalad run was recorded. This action is committed by DataLad under the original commit message of the run command, and looks just like the previous datalad run commit. Two cool tools that go beyond the git log (manual) are the datalad diff (manual) and git diff (manual) 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[1]): please use ‘datalad diff –from main –to HEAD~1’ datalad diff --to HEAD~1 modified: recordings/podcasts.tsv (file) This indeed shows the output file as “modified”. However, we do not know what exactly changed. This is a task for git diff (get out of the diff view by pressing q): git diff HEAD~1 diff --git a/recordings/podcasts.tsv b/recordings/podcasts.tsv index f691b53..d77891d 100644 --- a/recordings/podcasts.tsv +++ b/recordings/podcasts.tsv @@ -1,3 +1,31 @@ +2017-06-09 How Digital Memory Is Shaping Our Future Abby Smith Rumsey +2017-06-09 Pace Layers Thinking Stewart Brand Paul Saffo +2017-06-09 Proof The Science of Booze Adam Rogers +2017-06-09 Seveneves at The Interval Neal Stephenson +2017-06-09 Talking with Robots about Architecture Jeffrey McGrew +2017-06-09 The Red Planet for Real Andy Weir +2017-07-03 Transforming Perception One Sense at a Time Kara Platoni +2017-08-01 How Climate Will Evolve Government and Society Kim Stanley Robinson +2017-09-01 Envisioning Deep Time Jonathon Keats +2017-10-01 Thinking Long term About the Evolving Global Challenge The Refugee Reality +2017-11-01 The Web In An Eye Blink Jason Scott +2017-12-01 Ideology in our Genes The Biological Basis for Political Traits Rose McDermott +2017-12-07 Can Democracy Survive the Internet Nathaniel Persily +2018-01-02 The New Deal You Don t Know Louis Hyman This output actually shows the precise changes between the contents created with the first version of the script and the second script with the bug fix. All of the files that are added after the second directory was queried as well are shown in the diff, preceded by a +. Quickly create a note about these two helpful commands in notes.txt: 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) 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 (manual) 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 --): use ‘git log main – recordings/podcasts.tsv’ git log -- recordings/podcasts.tsv commit 08120c38✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [DATALAD RUNCMD] create a list of podcast titles === Do not change lines below === { "chain": ["e37c9fc9✂SHA1"], "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv", "dsid": "e3e70682-c209-4cac-629f-6fbed82c07cd", "exit": 0, "extra_inputs": [], "inputs": [], "outputs": [], "pwd": "." } ^^^ Do not change lines above ^^^ commit e37c9fc9✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [DATALAD RUNCMD] create a list of podcast titles === Do not change lines below === { "chain": [], "cmd": "bash code/list_titles.sh > recordings/podcasts.tsv", "dsid": "e3e70682-c209-4cac-629f-6fbed82c07cd", "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) Footnotes [1] The section Back and forth in time will elaborate more on common Git commands and terminology. ← 2.1. Keeping track 2.3. Input and output → v: latest The DataLad Handbook Table of Contents 2.3. Input and output 2.3.1. If outputs already exist… 2.3.2. Save yourself the preparation time 2.3.3. Placeholders 2.3.4. Dry-running your run call Related Topics Documentation overview Basics 2. DataLad, run! Previous: 2.2. DataLad, rerun! Next: 2.4. Clean desk Quick search 2.3. Input and output In the previous two sections, you created a simple .tsv file of all speakers and talk titles in the longnow/ podcasts subdataset, and you have re-executed a datalad run (manual) command after a bug-fix in your script. But these previous datalad run and datalad rerun (manual) 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 are 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. 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. Tool installation 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 convert: Unable to open file
(recordings/longnow/.datalad/feed_metadata/logo_salt.jpg) [No such file or directory].

[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' run(error): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] 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 cannot 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[1]. 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 (manual), and we did not datalad get (manual) 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. 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" get(ok): recordings/longnow/.datalad/feed_metadata/logo_salt.jpg (file) [from web...] [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] add(ok): recordings/salt_logo_small.jpg (file) save(ok): . (dataset) 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 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. What if there are several inputs? 2.3.1. If outputs already exist… Good news! Here is something that is easier on Windows 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 cannot use a datalad rerun for this: if we want to change the dimension option in the command, we have to define a new datalad run command. To establish best-practices, let's specify the input even though it is already present: datalad run -m "Resize logo for slides" \ --input "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \ "convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg recordings/salt_logo_small.jpg" # or shorter: datalad run -m "Resize logo for slides" \ -i "recordings/longnow/.datalad/feed_metadata/logo_salt.jpg" \ "convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg recordings/salt_logo_small.jpg" [INFO] == Command start (output follows) ===== convert convert: Unable to open file (recordings/salt_logo_small.jpg) [Permission denied]. [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' run(error): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 450x450 recordings/longn...] Oh wtf… What is it now? A quick glimpse into the error message shows a different error than before: The tool complains that it is "unable to open" the image, because the "Permission [is] denied". We have not seen anything like this before, and we need to turn to our lecturer for help. Confused about what we might have done wrong, we raise our hand to ask the instructor. Knowingly, she smiles, and tells you about how DataLad protects content given to it: "Content in your DataLad dataset is protected by git-annex from accidental changes" our instructor begins. "Wait!" we interrupt. "First off, that wasn't accidental. And second, I was told this course does not have git-annex-101 as a prerequisite?" "Yes, hear me out" she says. "I promise you two different solutions at the end of this explanation, and the concept behind this is quite relevant". DataLad usually gives content to git-annex to store and track. git-annex, let's just say, takes this task really seriously. One of its features that you have just experienced is that it locks content. If files are locked down, their content cannot 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 (manual). Let us check out what it does: What happens if I run this on Windows? 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 (manual) on this: datalad status modified: recordings/salt_logo_small.jpg (file) “Ah, do not mind that for now”, our instructor says, and with a wink she continues: “We’ll talk about symlinks and object trees a while later”. You are not really sure whether that’s a good thing, but you have a task to focus on. Hastily, you run the command right from the terminal: convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_salt.jpg recordings/salt_logo_small.jpg Hey, no permission denied error! You note that the instructor still stands right next to you. “Sooo… now what do I do to lock the file again?” you ask. “Well… what you just did there was quite suboptimal. Didn’t you want to use datalad run? But, anyway, in order to lock the file again, you would need to run a datalad save (manual).” datalad save -m "resized picture by hand" add(ok): recordings/salt_logo_small.jpg (file) save(ok): . (dataset) “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. But what if I have a lot of outputs? 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: Wait, would I need to specify outputs, too? datalad run -m "Resize logo for slides" \ --input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \ --output "recordings/interval_logo_small.jpg" \ "convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg recordings/interval_logo_small.jpg" # or shorter: datalad run -m "Resize logo for slides"

\ -i "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \ -o "recordings/interval_logo_small.jpg" \ "convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg recordings/interval_logo_small.jpg" get(ok): recordings/longnow/.datalad/feed_metadata/logo_interval.jpg (file) [from web...] [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 450x450 recordings/longn...] add(ok): recordings/interval_logo_small.jpg (file) save(ok): . (dataset) 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 2.3.2. Save yourself the preparation time Its generally good practice to specify --input and --output even if your input files are already retrieved and your output files unlocked – it makes sure that a recomputation can succeed, even if inputs are not yet retrieved, or if output needs to be unlocked. However, the internal preparation steps of checking that inputs exist or that outputs are unlocked can take a bit of time, especially if it involves checking a large number of files. If you want to avoid the expense of unnecessary preparation steps you can make use of the --assume-ready argument of datalad run. Depending on whether your inputs are already retrieved, your outputs

already unlocked (or not needed to be unlocked), or both, specify --assume-ready with the argument inputs, outputs or both and save yourself a few seconds, without sacrificing the ability to rerun your command under conditions in which the preparation would be necessary. 2.3.3. Placeholders Just after writing the note, you had to relax your fingers a bit. “Man, this was so much typing. Not only did I need to specify the inputs and outputs, I also had to repeat all of these lengthy paths in the command line call…” you think. There is a neat little trick to spare you half of this typing effort, though: Placeholders for inputs and outputs. This is how it works: Instead of running datalad run -m "Resize logo for slides" \ --input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \ --output "recordings/interval_logo_small.jpg" \ "convert -resize 450x450 recordings/longnow/.datalad/feed_metadata/logo_interval.jpg recordings/interval_logo_small.jpg" you could shorten this to datalad run -m "Resize logo for slides" \ --input "recordings/longnow/.datalad/feed_metadata/logo_interval.jpg" \ --output "recordings/interval_logo_small.jpg" \ "convert -resize 450x450 {inputs} {outputs}" The placeholder {inputs} will expand to the path given as --input, and the placeholder {outputs} will expand to the path given as --output. This means instead of writing the full paths in the command, you can simply reuse the --input and --output specification done before. What if I have multiple inputs or outputs? … wait, what if I need a curly bracket in my ‘datalad run’ call? 2.3.4. Dry-running your run call datalad run commands can become confusing and long, especially when you make heavy use of placeholders or wrap a complex bash commands. To better anticipate what you will be running, or help debug a failed command, you can make use of the --dry-run flag of datalad run. This option needs a mode specification (--dry-run=basic or dry-run=command), followed by the run command you want to execute, and it will decipher the commands elements: The mode command will display the command that is about to be ran. The mode basic will report a few important details about the execution: Apart from displaying the command that will be ran, you will learn where the command runs, what its inputs are (helpful if your --input specification includes a globbing term), and what its outputs are. [1] 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. ← 2.2. DataLad, rerun! 2.4. Clean desk → v: latest Table of Contents 2.4. Clean desk 2.4.1. Oh for f**** sake… run is “impossible”? Related Topics Documentation overview Basics 2. DataLad, run! Previous: 2.3. Input and output

Next: 2.5. Summary Quick search 2.4. Clean desk Just now you realize that you need to fit both logos onto the same slide. “Ah, damn, I might then really need to have them 400 by 400 pixel to fit”, you think. “Good that I know how to not run into the permission denied errors anymore!” Therefore, we need to do the datalad run (manual) 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] 2.4.1. 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 (manual) 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 (manual). Remember the section Populate a dataset: 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) 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" unlock(ok): recordings/interval_logo_small.jpg (file) [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] add(ok): recordings/interval_logo_small.jpg (file) save(ok): . (dataset) 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" unlock(ok): recordings/salt_logo_small.jpg (file) [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] add(ok): recordings/salt_logo_small.jpg (file) save(ok): . (dataset) With this flag, DataLad considers the specification of inputs and outputs to be “explicit”. It does not warn if the repository is dirty, but importantly, it only saves modifications to the listed outputs (which is a problem in the vast amount of cases where one does not exactly know which outputs are produced). Put explicit first! The --explicit flag has to be given anywhere prior to the command that should be run – the command needs to be the last element of a datalad run call. A datalad status will show that your previously modified notes.txt is still modified: $ datalad status modified: notes.txt (file) Add an additional note on the --explicit flag, and finally save your changes to notes.txt. $ cat << EOT >> notes.txt A suboptimal alternative is the --explicit flag, used to record only those changes done to the files listed with --output flags. EOT $ datalad save -m "add note on clean datasets" add(ok): notes.txt (file) save(ok): . (dataset) To conclude this section on datalad run, take a look at the last datalad run commit to see a run record with more content: $ git log -p -n 2 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [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": "e3e70682-c209-4cac-629f-6fbed82c07cd", "exit": 0, "extra_inputs": [], "inputs": ["recordings/longnow/.datalad/feed_metadata/logo_salt.jpg"], "outputs": ["recordings/salt_logo_small.jpg"], "pwd": "." } ^^^ Do not change lines above ^^^ diff

--git a/recordings/salt_logo_small.jpg b/recordings/salt_logo_small.jpg index 0985399..d90c601 120000 --- a/recordings/salt_logo_small.jpg +++ b/recordings/salt_logo_small.jpg @@ -1 +1 @@ ← 2.3. Input and output 2.5. Summary → v: latest The DataLad Handbook Table of Contents 2.4. Clean desk 2.4.1. Oh for f**** sake… run is “impossible”? Related Topics Documentation overview Basics 2. DataLad, run! Previous: 2.3. Input and output Next: 2.5. Summary Quick search 2.4. Clean desk Just now you realize that you need to fit both logos onto the same slide. “Ah, damn, I might then really need to have them 400 by 400 pixel to fit”, you think. “Good that I know how to not run into the permission denied errors anymore!” Therefore, we need to do the datalad run (manual) 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] 2.4.1. 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 (manual) 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 (manual). Remember the section Populate a dataset: 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) 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" unlock(ok): recordings/interval_logo_small.jpg (file) [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] add(ok): recordings/interval_logo_small.jpg (file) save(ok): . (dataset) 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" unlock(ok): recordings/salt_logo_small.jpg (file) [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows) ===== run(ok): /home/me/dl-101/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] add(ok): recordings/salt_logo_small.jpg (file) save(ok): . (dataset) With this flag, DataLad considers the specification of inputs and outputs to be “explicit”. It does not warn if the repository is dirty, but importantly, it only saves modifications to the listed outputs (which is a problem in the vast amount of cases where one does not exactly know which outputs are produced). Put explicit first! The --explicit flag has to be given anywhere prior to the command that should be run – the command needs to be the last element of a datalad run call. A datalad status will show that your previously modified notes.txt is still modified: datalad status modified: notes.txt (file) Add an additional note on the --explicit flag, and finally save your changes to notes.txt. cat << EOT >> notes.txt A suboptimal alternative is the --explicit flag, used to record only those changes done to the files listed with --output flags. EOT datalad save -m "add note on clean datasets" add(ok): notes.txt (file) save(ok): . (dataset) To conclude this section on datalad run, take a look at the last datalad run commit to see a run record with more content: git log -p -n 2 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 [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": "e3e70682-c209-4cac-629f-6fbed82c07cd", "exit": 0, "extra_inputs": [], "inputs":
["recordings/longnow/.datalad/feed_metadata/logo_salt.jpg"], "outputs":
["recordings/salt_logo_small.jpg"], "pwd": "." } ^^^ Do not change lines above ^^^ diff --git a/recordings/salt_logo_small.jpg b/recordings/salt_logo_small.jpg index 0985399..d90c601 120000 --- a/recordings/salt_logo_small.jpg +++ b/recordings/salt_logo_small.jpg @@ -1 +1 @@ ← 2.3. Input and output 2.5. Summary → v: latest The DataLad Handbook Table of Contents 2.5. Summary 2.5.1. Now what can I do with that? 2.5.2. Further reading Related Topics Documentation overview Basics 2. DataLad, run! Previous: 2.4. Clean desk Next: 3. Under the hood: git-annex Quick search 2.5. Summary In the last four sections, we demonstrated how to create a proper datalad run (manual) 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 (manual), 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 (manual) prior to command execution. Anything specified as output will be unlocked prior to modifications. It is good practice to specify input and output to ensure that a datalad rerun works, and to capture the relevant elements of a computation in a machine-readable record. If you want to spare yourself preparation time in case everything is already retrieved and unlocked, you can use --assume-ready {input|output|both} to skip a check on whether inputs are already present or outputs already unlocked. Schematic illustration of datalad run. Fig. 2.1 Overview of datalad run. 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 (manual) 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. 2.5.1. Now what can I 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 (manual) to unlock this content. The next section will elaborate on this a bit more. 2.5.2. Further reading The chapter on datalad run provided an almost complete feature overview of the command. If you want, you can extend this knowledge with computational environments and datalad containers-run (manual) in chapter Computational reproducibility with software containers. In addition, you can read up on other forms of computing usecases - for example, how to use datalad run in interactive computing environments such as Jupyter Notebooks. ← 2.4. Clean desk 3. Under the hood: git-annex → v: latest Related Topics Documentation overview Basics 3. Under the hood: git-annex Previous: 3. Under the hood: git-annex Next: 3.2. Data integrity Quick search 3.1. Data safety Later in the day, after seeing and solving so many DataLad error messages, you fall tired into your bed. Just as you are about to fall asleep, a thought crosses your mind: “I now know that tracked content in a dataset is protected by git-annex. Whenever tracked contents are saved, they get locked and should not be modifiable. But… what about the notes that I have been taking since the first day? Should I not need to unlock them before I can modify them? And also the script! I was able to modify this despite giving it to DataLad to track, with no permission denied errors whatsoever! How does that work?” This night, though, your question stays unanswered and you fall into a restless sleep filled with bad dreams about “permission denied” errors. The next day you are the first student in your lecturer’s office hours. “Oh, you are 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. It is because of this configuration that we can modify notes.txt without unlocking its content first. The second commit message in our datasets history summarizes this (outputs are shortened): $ git log --reverse --oneline 4ce681d [DATALAD] new dataset e0ff3a7 Instruct annex to add text files to Git b40316a add books on Python and Unix to read later a875e49 add reference book about git 59ac8d3 add beginners guide on bash 874d766 Add notes on datalad create e310b46 add note on datalad save 3c016f7 [DATALAD] Added subdataset 87609a3 Add note on datalad clone Instead of giving text files such as your notes or your script to git-annex, the dataset stores it in Git. But what does it mean if files are in Git instead of git-annex? Well, procedurally it means that everything that is stored in git-annex is content-locked, and everything that is stored in Git is not. You can modify content stored in Git straight away, without unlocking it first. Fig. 3.1 A simplified overview of the tools that manage data in your dataset. That's easy enough, and illustrated in Fig. 3.1. "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[1]." "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 (manual) 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. And hosting sites for projects using Git, such as GitHub or GitLab also do not allow files larger than a few dozen MB of size. For now, we have solved the mystery of why text files can be modified without unlocking, and this is a small improvement in the vast amount of questions that have piled up in our curious minds. Essentially, git-annex protects your data from accidental modifications and thus keeps it safe. datalad run commands mitigate any technical complexity of this completely if -o/--output is specified properly, and datalad unlock (manual) commands can be used to unlock content "by hand" if modifications are performed outside of a datalad run. But there comes the second, tricky part: There are ways to get rid of locking and unlocking within git-annex, using so-called adjusted branches. This functionality is dependent on the git-annex version one has installed, the git-annex version of the repository, and a use-case dependent comparison of the pros and cons. On Windows systems, this adjusted mode is even the only mode of operation. In later

sections we will see how to use this feature. The next lecture, in any way, will guide us deeper into git-annex, and improve our understanding a slight bit further. Footnotes [1] If you cannot wait to read about .gitattributes and other configuration files, jump ahead to chapter Tuning datasets to your needs, starting with section DIY configurations. ← 3. Under the hood: git-annex 3.2. Data integrity → v: latest The DataLad Handbook Table of Contents 3.2. Data integrity 3.2.1. Broken symlinks Related Topics Documentation overview Basics 3. Under the hood: git-annex Previous: 3.1. Data safety Next: 4. Collaboration Quick search 3.2. Data integrity So far, we mastered quite a number of challenges: Creating and populating a dataset with large and small files, modifying content and saving the changes to history, installing datasets, even as subdatasets within datasets, recording the impact of commands on a dataset with the datalad run (manual) and datalad rerun (manual) commands, and capturing plenty of provenance on the way. We further noticed that when we modified content in notes.txt or list_titles.sh, the modified content was in a text file. We learned that this precise type of file, in conjunction with the initial configuration template text2git we gave to datalad create (manual), is meaningful: As the text file is stored in Git and not git-annex, no content unlocking is necessary. As we saw within the demonstrations of datalad run, modifying content of non-text files, such as .jpgs, typically requires the additional step of unlocking file content, either by hand with the datalad unlock (manual) command, or within datalad run using the -o/--output flag. There is one detail about DataLad datasets that we have not covered yet. It is a crucial component 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: Dataset directories look different on Windows # in the root of DataLad-101 cd books tree . ├── bash_guide.pdf -> ../.git/annex/objects/WF/Gq/✂/MD5E-s1198170--0ab2c121✂MD5.pdf ├── byte-of-python.pdf -> ../.git/annex/objects/xF/42/✂/MD5E-s4161086--c832fc13✂MD5.pdf ├── progit.pdf -> ../.git/annex/objects/G6/Gj/✂/MD5E-s12465653--05cd7ed5✂MD5.pdf └── TLCL.pdf -> ../.git/annex/objects/jf/3M/✂/MD5E-s2120211--06d1efcb✂MD5.pdf 0 directories, 4 files If you do not know what you are looking at, this looks weird, if not worse: intimidating, wrong, or broken. First of all: no, it is all fine. But let's start with the basics of what is displayed here to understand it. The small -> symbol connecting one path (the book's name) to another path (the weird sequence of

characters ending in .pdf) is what is called a symbolic link (short: symlink) or softlink. It is a term for any file that contains a reference to another file or directory as a relative path or absolute path. If you use Windows, you are familiar with a related, although more basic concept: a shortcut. This means that the files that are in the locations in which you saved content and are named as you named your files (e.g., TLCL.pdf), do not actually contain your files' content: they just point to the place where the actual file content resides. This sounds weird, and like an unnecessary complication of things. But we will get to why this is relevant and useful shortly. First, however, where exactly are the contents of the files you created or saved? The start of the link path is ../.git. The section Create a dataset contained a note that strongly advised that you to not tamper with (or in the worst case, delete) the .git repository in the root of any dataset. One reason why you should not do this is because this .git directory is where all of your file content is actually stored. But why is that? We have to talk a bit git-annex now in order to understand it. 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 file size 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. File content management on Windows (adjusted mode) For a demonstration that this file path is not complete gibberish, take the target path of any of the book's symlinks and open it, for example with evince <path>, or any other PDF reader in exchange for evince: evince ../.git/annex/objects/jf/3M/</MD5E-s2120211--06d1efcb<MD5.pdf Even though the path looks cryptic, it works and opens the file. Whenever you use a command like evince TLCL.pdf, internally, programs will follow the same cryptic symlink like the one you have just opened. But why does this symlink-ing happen? Up until now, it still seems like a very unnecessary, superfluous thing to do, right? The resulting symlinks that look like your files but only point to the actual content in .git/annex/objects are small in size. An ls -lh reveals that all of these symlinks have roughly the same, small size of ~130 Bytes: ls -lh total 16K lrwxrwxrwx 1 elena elena 131 2019-06-18 16:13 bash_guide.pdf -> ../.git/annex/objects/WF/Gq/</MD5E-s1198170--0ab2c121<MD5.pdf lrwxrwxrwx 1 elena elena 131 2019-06-18 16:13 byte-of-

python.pdf -> ../.git/annex/objects/xF/42/</MD5E-s4161086--c832fc13<MD5.pdf lrwxrwxrwx 1 elena elena 133 2019-06-18 16:13 progit.pdf -> ../.git/annex/objects/G6/Gj/</MD5E-s12465653--05cd7ed5<MD5.pdf lrwxrwxrwx 1 elena elena 131 2019-06-18 16:13 TLCL.pdf -> ../.git/annex/objects/jf/3M/</MD5E-s2120211--06d1efcb<MD5.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 Find-out-more about the object tree. 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. Compared to copying and deleting huge data files, small symlinks can be written very very fast, for example, when switching dataset versions, or branches. Speedy branch switches Switching branches fast, even when they track vasts amounts of data, lets you work with data with the same routines as in software development. This leads to a few conclusions: The first is that you should not be worried to see cryptic looking symlinks in your repository – this is how it should look. You can read the find-out-more on why these paths look so weird and what all of this has to do with data integrity, if you want to. It's additional information that can help to establish trust in that your data are safely stored and tracked, and understanding more about the object tree and knowing bits of the git-annex basics can make you more confident in working with your datasets. The second is that it should now be clear to you why the .git directory should not be deleted or in any way modified by hand. This place is where your data are stored, and you can trust git-annex to be better able to work with the paths in the object tree than you or any other human are. Lastly, understanding that annexed files in your dataset are symlinked will be helpful to understand how common file system operations such as moving, renaming, or copying content translate to dataset modifications in certain situations. Later in this book, the section Miscellaneous file system operations will take a closer look at that. Data integrity and annex keys 3.2.1. 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 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 we explored the file hierarchy in an installed subdataset that contained many annexed mp3 files. Upon the initial datalad clone (manual), the annexed files were not present locally. Instead, their symlinks (stored in Git) existed and allowed to explore which file's contents could be retrieved. These symlinks point to nothing, though, as the content isn't yet present locally, and are thus broken. This state, however, is not problematic at all. Once the content is retrieved via datalad get (manual), the symlink is functional again. Nevertheless, it may be important to know that some tools that you would expect to work in a dataset with not yet retrieved file contents can encounter unintuitive problems. Some file managers (e.g., OSX's Finder) may not display broken symlinks. In these cases, it will be impossible to browse and explore the file hierarchy of not-yet-retrieved files with the file manager. You can make sure to always be able to see the file hierarchy in two separate ways: Upgrade your file manager to display file types in DataLad datasets (e.g., the git-annex-turtle extension for Finder), or use the DataLad Gooey to browse datasets. Alternatively, use the ls command in a terminal instead of a file manager GUI. Other tools may be more more specialized, smaller, or domain-specific, and may fail to correctly work with broken symlinks, or display unhelpful error messages when handling them, or require additional flags to modify their behavior. When encountering unexpected behavior or failures, try to keep in mind that a dataset without retrieved content appears to be a pile of broken symlinks to a range of tools, consult a tools documentation with regard to symlinks, and check whether data retrieval fixes persisting problems. A last special case on symlinks exists if you are using DataLad on the Windows Subsystem for Linux. If so, please take a look into the Windows Wit below. Accessing symlinked files from your Windows system Finally, if you are still in the books/ directory, go back into the root of the superdataset. cd ../ ← 3.1. Data safety 4. Collaboration → v: latest Related Topics Documentation overview Basics 4. Collaboration Previous: 4. Collaboration Next: 4.2. Where's Waldo? Quick search 4.1. Looking without touching Only now, several weeks into the DataLad-101 course does your room mate realize that he has enrolled in the course as well, but has not yet attended at all. "Oh man, can you help me catch up?" he asks you one day. "Sharing just your notes would

be really cool for a start already!" "Sure thing", you say, and decide that it's probably best if he gets all of the DataLad-101 course dataset. Sharing datasets was something you wanted to look into soon, anyway. This is one exciting aspect of DataLad datasets that has yet been missing from this course: How does one share a dataset? In this section, we will cover the simplest way of sharing a dataset: on a local or shared file system, via an installation with a path as a source. More on public data sharing Interested in sharing datasets publicly? Read this chapter to get a feel for all relevant basic concepts of sharing datasets. Afterwards, head over to chapter Third party infrastructure to find out how to share a dataset on third-party infrastructure. In this scenario multiple people can access the very same files at the same time, often on the same machine (e.g., a shared workstation, or a server that people can "SSH" into). You might think: "What do I need DataLad for, if everyone can already access everything?" However, universal, unrestricted access can easily lead to chaos. DataLad can help facilitate collaboration without requiring ultimate trust and reliability of all participants. Essentially, with a shared dataset, collaborators can see and use your dataset without any danger of undesired, or uncontrolled modification. 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 are 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 cannot 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 section Install datasets. 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, subdatasets, run records – 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 are 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 (manual), and provide a path to your original dataset: $ cd mock_user $ datalad clone --description "DataLad-101 in 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. 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--0ab2c121<MD5.pdf │ ├── byte-of-python.pdf -> ../.git/annex/objects/xF/42/</MD5E-s4161086--c832fc13<MD5.pdf │ ├── progit.pdf -> ../.git/annex/objects/G6/Gj/</MD5E-s12465653--05cd7ed5<MD5.pdf │ └── TLCL.pdf -> ../.git/annex/objects/jf/3M/</MD5E-s2120211--06d1efcb<MD5.pdf ├── code │ └── list_titles.sh ├── notes.txt └── recordings ├── interval_logo_small.jpg -> ../.git/annex/objects/pw/Mf/</MD5E-s70348--4b2ec0db<MD5.jpg ├── longnow ├── podcasts.tsv └── salt_logo_small.jpg -> ../.git/annex/objects/fZ/wg/</MD5E-s76402--87da732f<MD5.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. 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] Remote origin not usable by git-annex; setting annex-ignore 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] Before we look further into datalad get (manual) and the -n/--no-data option, 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--0ab2c121✂MD5.pdf │ ├── byte-of-python.pdf -> ../.git/annex/objects/xF/42/✂/MD5E-s4161086--c832fc13✂MD5.pdf │ ├── progit.pdf -> ../.git/annex/objects/G6/Gj/✂/MD5E-s12465653--05cd7ed5✂MD5.pdf │ └── TLCL.pdf -> ../.git/annex/objects/jf/3M/✂/MD5E-s2120211--06d1efcb✂MD5.pdf ├── code │ └── list_titles.sh ├── notes.txt └── recordings ├── interval_logo_small.jpg -> ../.git/annex/objects/pw/Mf/✂/MD5E-s70348--4b2ec0db✂MD5.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--c723d53d✂MD5.mp3 │ │ ├── 2017_06_09__Pace_Layers_Thinking__Stewart_Brand__Paul_Saffo.mp3 -> ../.git/annex/objects/Qk/9M/✂/MD5E-s112801659--00a42a1a✂MD5.mp3 │ │ ├── 2017_06_09__Proof__The_Science_of_Booze__Adam_Rogers.mp3 -> ../.git/annex/objects/FP/96/✂/MD5E-s60091960--6e48eceb✂MD5.mp3 │ │ ├── 2017_06_09__Seveneves_at_The_Interval__Neal_Stephenson.mp3 -

> ../.git/annex/objects/Wf/5Q/🖋/MD5E-s66431897--aff90c83🖋MD5.mp3 │ │ ├──
2017_06_09__Talking_with_Robots_about_Architecture__Jeffrey_McGrew.mp3 -
> ../.git/annex/objects/Fj/9V/🖋/MD5E-s61491081--c4e88ea0🖋MD5.mp3 │ │ ├──
2017_06_09__The_Red_Planet_for_Real__Andy_Weir.mp3 -
> ../.git/annex/objects/xq/Q3/🖋/MD5E-s136924472--0d107210🖋MD5.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, datalad get operates 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), datalad 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 datalad 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 datalad 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 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[1]: The --recursion-limit option let's you specify how many levels of subdatasets should be installed together with the first subdataset: $ datalad get -n -r --recursion-limit 1 <subds> 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 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) Get a clone A dataset that is installed from an existing source, e.g., a path or URL, is the DataLad equivalent of a clone in Git. Footnotes [1] Another alternative to a recursion limit to datalad get -n -r is a dataset configuration that specifies subdatasets that should not be cloned recursively, unless explicitly given to the command with a path. With this configuration, a superdataset's maintainer can safeguard users and prevent potentially large amounts of subdatasets to be cloned. You can learn more about this configuration in the section More on DIY configurations. ← 4. Collaboration 4.2. Where's Waldo? → 4.2. Where's Waldo? So far, you and your room mate have created a copy of the DataLad-101 dataset on the same file system but a different place by installing it from a path. You have observed that the -r/--recursive option needs to be given to datalad get [-n/--no-data] (manual) 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. 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 another PDF, let's query git-annex where its content is stored: $ # navigate back into the clone of DataLad-101 $ cd ../mock_user/DataLad-101 $ git annex whereis books/TLCL.pdf whereis books/TLCL.pdf (1 copy) 0c450dc0-c48c-4057-a231-e2654b689600 -- me@appveyor-vm:~/dl-101/DataLad-101 [origin] ok Oh, another cryptic character sequence - this time however not a symlink, but an annex UUID. "That's hard to read – what is it?" your room mate asks. You can recognize a path to the dataset on your computer, prefixed with the user and hostname of your computer. "This", you exclaim, excited about your own realization, "is my dataset's location I'm sharing it from!" What is this location, and what if I provided a description? 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 UUID to identify these locations. Every copy of a dataset will get a UUID as a unique identifier. 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 cannot 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 (manual) 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 0c450dc0-c48c-4057-a231-e2654b689600 -- me@appveyor-vm:~/dl-101/DataLad-101 [origin] web: https://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. We can also see a report of the source that git-annex uses to retrieve the content from if we look at the very end of the get summary. $ datalad get books/TLCL.pdf $ datalad get books/bash_guide.pdf get(ok): books/TLCL.pdf (file) [from origin...] get(ok): books/bash_guide.pdf (file) [from origin...] Both of these files were retrieved "from origin...". Origin is Git terminology for "from where the dataset was copied from" – origin therefore is the original DataLad-101 dataset from which file content can be retrieved from very fast. If your roommate did not have access to the same file system or you deleted your DataLad-101 dataset, this output would look differently.
The datalad get command would fail on the TLCL.pdf book without a known second source, and bash_guide.pdf would be retrieved "from web..." - the registered second source, its original download URL. Let's see a retrieval from web in action for another file. The .mp3 files in the longnow seminar series have registered web URLs[1]. $ # navigate into the subdirectory $ cd recordings/longnow $ git annex whereis Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.mp3 $ datalad get 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 (2 copies) 00000000-0000-0000-0000-000000000001 -- web ✂UUID✂ -- mih@medusa:/tmp/seminars-on-longterm-thinking web: http://podcast.longnow.org/salt/redirect/salt-020031114-eno-podcast.mp3 ok get(ok): Long_Now__Seminars_About_Long_term_Thinking/2003_11_15__Brian_Eno__The_Long_Now.mp3 (file) [from web...] As you can see at the end of the get result, the files has been retrieved "from web...". Quite useful, this provenance, right? Let's add a note on the git annex whereis command. Again, do this in the original DataLad-101 directory, and do not forget to save it. $ # navigate back: $ cd ../../../../DataLad-101 $ # write the note $ cat << EOT >> notes.txt The command "git annex whereis PATH" lists the repositories that have the file content of an annexed file. When using "datalad get" to retrieve file content, those repositories will be queried. EOT $ datalad status modified: notes.txt (file) $ datalad save -m "add note on git annex whereis" add(ok): notes.txt (file) save(ok): . (dataset) Footnotes [1] 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

cannot retrieve content from this location. However, somebody else (for example, the user mih) could. ← 4.1. Looking without touching 4.3. Retrace and reenact → v: latest The DataLad Handbook Related Topics Documentation overview Basics 4. Collaboration Previous: 4.2. Where's Waldo? Next: 4.4. Stay up to date Quick search 4.3. Retrace and reenact "Thanks a lot for sharing your dataset with me! This is super helpful. I'm sure I'll catch up in no time!", your room mate says confidently. "How far did you get with the DataLad commands yet?" he asks at last. "Mhh, I think the last big one was datalad run (manual). 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 (manual) these commits easily. This is what you want to try now. You begin to think about which datalad run commit would be the most useful one to take a look at. The creation of podcasts.tsv was a bit dull – at this point in time, you didn't yet know about --input and --output arguments, and the resulting output is present anyway because text files like this .tsv file are stored in Git. However, one of the attempts to resize a picture could be useful. The input, the podcast logos, is not yet retrieved, nor is the resulting, resized image. "Let's go for this!", you say, and drag your confused room mate to the computer screen. First of all, find the commit shasum of the command you want to run by taking a look into the history of the dataset (in the shared dataset): # navigate into the shared copy cd ../mock_user/DataLad-101 # lets view the history git log --oneline -n 10 41fdaab add note on clean datasets fd4a25e [DATALAD RUNCMD] Resize logo for slides 08f2efb [DATALAD RUNCMD] Resize logo for slides 87b4f80 add additional notes on run options d0c060f [DATALAD RUNCMD] convert -resize 450x450 recordings/longn... 5227d93 resized picture by hand 6686234 [DATALAD RUNCMD] convert -resize 400x400 recordings/longn... 4b7a0a0 add note on basic datalad run and datalad rerun cdedbc3 add note datalad and git diff 08120c3 [DATALAD RUNCMD] create a list of podcast titles Ah, there it is, the second most recent commit. Just as already done in section DataLad, rerun!, take this shasum and plug it into a datalad rerun command: datalad rerun fd4a25e6✂SHA1 [INFO] run commit fd4a25e; (Resize logo for s...) get(ok): recordings/longnow/.datalad/feed_metadata/logo_salt.jpg (file) [from web...] run.remove(ok): recordings/salt_logo_small.jpg (file) [Removed file] [INFO] == Command start (output follows) ===== [INFO] == Command exit (modification check follows)

===== run(ok): /home/me/dl-101/mock_user/DataLad-101 (dataset) [convert -resize 400x400 recordings/longn...] add(ok): recordings/salt_logo_small.jpg (file) action summary: add (ok: 1) get (notneeded: 1, ok: 1) run (ok: 1) run.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”, but is reportedly “removed” prior to the successful rerun. 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. ← 4.2. Where’s Waldo? 4.4. Stay up to date → v: latest Related Topics Documentation overview Basics 4. Collaboration Previous: 4.3. Retrace and reenact Next: 4.5. Networking Quick search 4.4. Stay up to date All of what you have seen about sharing dataset was really cool, and for the most part also surprisingly intuitive. datalad run (manual) commands or file retrieval worked exactly as you imagined it to work, and you begin to think that slowly but steadily you are 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 are 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 (manual), datalad get (manual), and git annex whereis (manual) into the original DataLad-101? This is a change that is not reflected in your “shared” installation in ../mock_user/DataLad-101: $ # Inside the installed copy, view the last 15 lines of notes.txt $ tail notes.txt should be specified with an -o/--output flag. Upon a run or rerun of the command, the contents of these files will get unlocked so that they can be modified. Important! If the dataset is not "clean" (a datalad status output is empty), datalad run will not work - you will have to save modifications present in your dataset. A suboptimal alternative is the --explicit flag, used to record only those changes done to the files listed with --output flags. But the original intention of sharing the dataset with your room mate was to give him access to your notes. How does he get the notes that you have added in the last two sections, for example? This installed copy of DataLad-101 knows its origin, i.e., the place it was installed from. Using this information, it can

query the original dataset whether any changes happened since the last time it checked, and if so, retrieve and integrate them. This is done with the datalad update --how merge (manual) command. $ datalad update --how merge merge(ok): . (dataset) [Merged origin/main] update.annex_merge(ok): . (dataset) [Merged annex branch] 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: $ # view the last 15 lines of notes.txt $ tail notes.txt Note that a recursive "datalad get" would install all further registered subdatasets underneath a subdataset, so a safer way to proceed is to set a decent --recursion-limit: "datalad get -n -r --recursion-limit 2 <subds>" The command "git annex whereis PATH" lists the repositories that have the file content of an annexed file. When using "datalad get" to retrieve file content, those repositories will be queried. Wohoo, the contents are here! Therefore, sharing DataLad datasets by installing them enables you to update the datasets content should the original datasets' content change – in only a single command. How cool is that?! Conclude this section by adding a note about updating a dataset to your own DataLad-101 dataset: $ # navigate back: $ cd ../../DataLad-101 $ # write the note $ cat << EOT >> notes.txt To update a shared dataset, run the command "datalad update --how 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) PS: You might wonder what a plain datalad update command with no option does. If you are a Git-user and know about branches and merging you can read the Note for Git-users. However, a thorough explanation and demonstration will be in the next section. Update internals datalad update is the DataLad equivalent of a git fetch (manual), datalad update --how merge is the DataLad equivalent of a git pull (manual). Upon a simple datalad update, the remote information is available on a branch separate from the main branch – in most cases this will be remotes/origin/main. You can git checkout (manual) this branch or run git diff (manual) to explore the changes and identify potential merge conflicts. ← 4.3. Retrace and reenact 4.5. Networking → 4.5. Networking To get a hang on the basics of sharing a dataset, you shared

your DataLad-101 dataset with your room mate on a common, local file system. Your lucky room mate now has your notes and can thus try to catch up to still pass the course. Moreover, though, he can also integrate all other notes or changes you make to your dataset, and stay up to date. This is because a DataLad dataset makes updating shared data a matter of a single datalad update --how merge (manual) command. But why does this need to be a one-way street? “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 cannot get mine.” Consider, for example, that your room mate might have googled about DataLad a bit. In the depths of the web, he might have found useful additional information, such a script on dataset nesting. Because he found this very helpful in understanding dataset nesting concepts, he decided to download it from GitHub, and saved it in the code/ directory. He does it using the DataLad command datalad download-url (manual) that you experienced in section Create a dataset 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/7e8e39b1/content/asciicast/seamless_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) Run a quick datalad status: $ datalad status nothing to save, working tree clean Nice, the datalad download-url command saved this download right into the history, and datalad status (manual) does not report unsaved modifications! We’ll show an excerpt of the last commit here[1]: $ git log -n 1 -p commit 5b6e19a5✂SHA1 Author: Elena Piscopia <elena@example.net> Date: Tue Jun 18 16:13:00 2019 +0000 Include nesting demo from datalad website diff --git a/code/nested_repos.sh b/code/nested_repos.sh new file mode 100644 index 0000000..f84c817 --- /dev/null +++ b/code/nested_repos.sh @@ -0,0 +1,59 @@ Suddenly, your room mate has a file change that you do not have. His dataset evolved. So how do we link back from the copy of the dataset to its origin, such that your room mate’s changes can be included in your dataset? How do we let the original dataset “know” about this copy your room

mate has? Do we need to install the installed dataset of our room mate as a copy again? No, luckily, it's simpler and less convoluted. What we have to do is to register a DataLad sibling: A reference to our room mate's dataset in our own, original dataset. Remote siblings Git repositories can configure clones of a dataset as remotes in order to fetch, pull, or push from and to them. A datalad sibling (manual) 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 (manual) command. 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. What if I mistyped the name or want to remove the sibling? 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 --how merge in the section Stay up to date. 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 --how 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 (which uses the default option --how fetch), 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 --how merge option. $ datalad update -s roommate 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 --how merge worked for your room mate), because there is only one other known location, though. This plain datalad update "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/main. To look inside of it you can git checkout BRANCHNAME (manual), 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: Please use 'datalad diff –from main –to remotes/roommate/main' $ datalad diff --to remotes/roommate/main 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 (manual) to show us what the differences in detail (note that it is a shortened excerpt, cut in the middle to reduce its length): Please use 'git diff main..remotes/roommate/main' $ git diff remotes/roommate/main 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 diff --git a/notes.txt b/notes.txt index 655be7d..ff02f68 100644 --- a/notes.txt +++ b/notes.txt @@ -59,3 +59,7 @@ The command "git annex whereis PATH" lists the repositories that have the file content of an annexed file. When using "datalad get" to retrieve file content, those repositories will be queried. +To update a shared dataset, run the command "datalad update --how 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 --how merge them to integrate them into your dataset. In technical terms you will “merge the branch remotes/roommate/main into main”. B

ut 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 --how merge -s roommate merge(ok): . (dataset) [Merged roommate/main] update.annex_merge(ok): . (dataset) [Merged annex branch] 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 6ae8e71 Merge remote-tracking branch 'roommate/main' 4bb5d39 add note about datalad update 5b6e19a Include nesting demo from datalad website adb4b5d add note on git annex whereis 1e73592 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 “main” is. For now, you are 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 --how 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/main EOT $ datalad save -m "Add note on adding siblings" add(ok): notes.txt (file) save(ok): . (dataset) Footnotes [1] As this example, simplistically, created a “pretend” room mate by only changing directories, not user accounts, the recorded Git identity of your “room mote” will, of course, be the same as yours. ← 4.4. Stay up to date 4.6. Summary → v: latest The DataLad Handbook Table of Contents 4.6. Summary 4.6.1. Now what can I do with that? Related Topics Documentation overview Basics 4. Collaboration Previous: 4.5. Networking Next: 5. Tuning datasets to your needs Quick search 4.6. Summary Together with your room mate you have just discovered how to share, update, and collaborate on a DataLad dataset on a shared file system. Thus, you have glimpsed into the principles and advantages of sharing a dataset with a simple example. To obtain a dataset, one can also use datalad clone (manual) 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 (manual): 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 (manual) will list all locations known to contain file content for a particular file. It is a very helpful command to find out where file content resides, and how many locations with copies exist. git-annex will try to retrieve file contents from those locations. If you want, you can

describe locations with the --description provided during a datalad create (manual). A shared copy of a dataset includes the datasets history. If well made, datalad run (manual) 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 (manual) command. This command will query the origin of the dataset for updates, and a datalad update --how 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 (manual) and git diff (manual) can subsequently help to find out what changes have been made in the sibling. 4.6.1. Now what can I 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 (manual)) 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 (manual) 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 --how 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. ← 4.5. Networking 5. Tuning datasets to your needs → v: latest