Introduction to Computing

Edited: J. Swift

March 25, 2020

Contents

List of Figures

Preface

This quick reference guide is meant to aid you in setting up and becoming familiar with the use of the command-line on your computer. It is geared particularly for students of Introduction to Data Science at the Thacher School. However, it may be useful for any student looking to learn the basics of setting up and using `python` on their computers.

Given the ubiquity of Mac computers on campus, this tutorial is aimed at those using a Mac OS X operating system. Though, given that it is UNIX at the core, there will be much overlap with setting up a Linux machine as well. Windows is not directly supported in this document.

Structure

There are only 3 chapters to this document. The first focuses on getting one familiar with the UNIX environment. The second chapter details how to set up a python installation and become familiar with some basic tools such as `spyder`—an integrated development environment (IDE) for python.

The third chapter then delves into the basic functionality of `git`. This is a standard utility on your Mac or Linux computers and we will explore the command line functionality of this powerful program in this chapter. Use of GUI based `git` interfaces such as *GitKraken*, will be left to the user to explore.

It should be fully appreciated that almost everything you will ever need to know about getting started with command-line computing and programming can be found on-line. You should not be shy using "google" to find answers to your problems. It will be very unlikely that you will encounter a problem for the first time at this level, and most problems you will encounter will also be solved and well-documented.

Uniqueness

Much of the information provided in this document is freely available on the internet. Many paragraphs were pilfered directly from websites, manual pages, and emails. Given the heterogeneous sourcing of information, there has been very little accounting of information provenance. Keep in mind this is an amalgamation and regurgitation of material rather than a unique work. Therefore, this document should be used as a reference manual and should not be reproduced for any reason.

Acknowledgements

Thanks to Amber Jain for the LaTeX book template*.

— J. Swift

*https://github.com/amberj/latex-book-template

1

Introduction to UNIX

"This is a quote and I don't know who said it."
– inspired by M.C. Escher

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment (*e.g.*, Mac OS X). However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available. While there seem to be ever more GUI programs for every task imaginable, using a GUI obfuscates the program machinery "under the hood." Since we will be primarily concerned with how your computer works. We will be avoiding GUIs whenever possible.

1.1 The Kernel

Technically speaking, the UNIX kernel *is* the operating system. It provides the basic full time software connection to the hardware. By full time, I mean that the kernel is always running while the computer is turned on. When the system boots up, the kernel is loaded. Likewise, the kernel is only exited when the computer is turned off.

The UNIX kernel is built specifically for a machine when it is installed. It has a record of all the pieces of hardware it needs to talk to and knows what languages they speak (how to turn switches on and off to get a desired result).

Thus, a kernel is not easily ported to another computer. Each individual computer will have its own tailor- made kernel. And if the computer's hardware configuration changes during its life, the kernel must be "rebuilt" (told about the new pieces of hardware).

However, though the connection between the kernel and the hardware is "hard-coded" to a specific machine, the connection between the user and the kernel is generic. That is the beauty of the UNIX kernel. From your perspective, regardless of how the kernel interacts with the hardware, no matter which UNIX computer you use, you will have the same kernel interface to work with. That is because the hardware is "hidden" by the kernel.

However, the task of hiding the hardware is a pretty much full time job for the kernel. As such, it does not have too much time to provide for a fancy user-friendly interface. Thus, though the kernel is much easier to talk to than the hardware, the language of the kernel is still pretty cryptic. Fortunately, the UNIX operating system has built in "shells" which wrap around the kernel and provide a much more user-friendly interface.

1.2 The Shell

In UNIX, the shell is a program that interprets commands and acts as an intermediary between the user and the inner workings of the operating system, providing a command-line interface (i.e., the shell prompt or command prompt). On most UNIX systems, there are several shells available. For the average user, they offer similar functionality, but each has different syntax and capabilities. Most shells fall within one of two classes: those descended from the Bourne shell (*i.e.*, `sh`), and those from the C Shell (i.e., `csh`). Nearly every UNIX system has these two shells installed, but may also have several others, *e.g.*, `bash`, `ksh`, `tcsh`, and `zsh`. To determine what options are available for your login shell (*i.e.*, your default shell), look at the file `/etc/shells` on your system.

Most shells double as interpreted programming languages. To automate tasks, you may write scripts containing built-in shell and UNIX commands. When you execute a script, the shell interprets these commands as if you had entered them from the command-line prompt. Compared to compiled programs, shell scripts are slow but easy to write and debug.

1.3 The Terminal Application

The Terminal is a program that runs a UNIX shell and it is included with all versions of Mac OS X. It is located in the Utilities folder within the Applications folder. When launched, it provides a line interface to control the underpinnings of the UNIX based operating system.

Open the Terminal application on your Mac by either clicking on the application directly in /Applications/Utilities, or searching for it with Spotlight. Once a Terminal is open you will see a command prompt likely followed by some information about your computer's name and what directory you are in (a directory is what you might call a folder if you were a less sophisticated user). You may type commands and issue them by pressing return.

1.4 Navigating your computer on the command line

File and directory paths in UNIX use the forward slash ("/") to separate directory names. For example, `/Users/jonswift/python` would designate the directory `python` that is in the directory `jonswift`, that exists in the directory `/Users`. The succession of directories is called a "path."

There are some simple UNIX commands that allow you to navigate the directory tree of your computer and to see where along the path you are at any time. These commands also have many many options. We will talk a little bit about options in this section, and will dive in deeper after this introduction.

`ls`

Lists the contents of a directory. Can be entered as is to list all the contents of the current directory, or it can include options and arguments.

`cd`

This command changes the current directory that you are in to a named directory. If entered with no argument, it will return you to your home directory.

pwd

The pwd utility writes the absolute pathname of the current working directory to the standard output.

whoami

The whoami utility displays your effective user ID as a name

who

The who utility displays a list of all users currently logged on, showing for each user the login name, tty name, the date and time of login, and hostname if not local.

The dot and double dot

When a period, or dot (" . ") is used along a path, it points to the directory that your are in, or the directory last specified along a path designation. It is used to be explicit about the path designation. For instance if you want to run a program called `myscript` that exists in your current directory and ensure that your computer will not execute another program of the same name along your computer's search path, you would type `./myscript`.

Another special UNIX character is the double period, or double dot (" .. "). This is used more frequently than the single period and specifies one directory backward from where you are, or one backward from the last directory specified along a path designation. For example, if you change directories into `Documents` by issuing the command `cd Documents`. To return to the directory from which you just came, you would simply type "`cd ..`".

1.5 Handling files and directories

cp

Copy a file or directory to a destination. If copying a directory, the `-r` option must be used.

mv

In its first form, the mv utility renames the file named by the source operand to the destination path named by the target operand. This form is assumed when the last operand does not name an already existing directory.

In its second form, `mv` moves each file named by a source operand to a destination file in the existing directory named by the directory operand. The destination path for each operand is the pathname produced by the concatenation of the last operand, a slash, and the final pathname component of the named file.

`cat`

The cat utility reads files sequentially, writing them to the standard output. The file operands are processed in command-line order. If file is a single dash ('-') or absent, cat reads from the standard input. If file is a UNIX domain socket, cat connects to it and then reads it until EOF.

`more` and `less`

Both `less` and `more` allow one to navigate an ASCII file from the command line in an easy and robust way. `less` is similar to `more`, but allows backward movement in the file as well as forward movement. Also, `less` does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like `vi` or `emacs`. `less` uses termcap (or terminfo on some systems), so it can run on a variety of terminals. There is even limited support for hardcopy terminals. (On a hardcopy terminal, lines which should be printed at the top of the screen are prefixed with a caret.)

Commands are based on both more and vi. Commands may be preceded by a decimal number, called N in the descriptions below. The number is used by some commands, as indicated.

`rm`

The `rm` utility attempts to remove the non-directory type files specified on the command line. If the permissions of the file do not permit writing, and the standard input device is a terminal, the user is prompted (on the standard error output) for confirmation.

`rmdir`

The `rmdir` utility removes the directory entry specified by each directory argument, provided it is empty.

Arguments are processed in the order given. In order to remove both a parent directory and a subdirectory of that parent, the subdirectory must be

specified first so the parent directory is empty when `rmdir` tries to remove it.

1.6 Higher level commands

`jobs`

The shell associates a job with each pipeline. It keeps a table of current jobs, printed by the jobs command, and assigns them small integer numbers. When a job is started asynchronously with '&', the shell prints a line which looks like:
`[1] 1234`
indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process ID was 1234.

`ps`

The `ps` utility displays a header line, followed by lines containing information about all of your processes that have controlling terminals.

`grep`

The `grep` utility searches any given input files, selecting lines that match one or more patterns. By default, a pattern matches an input line if the regular expression (RE) in the pattern matches the input line without its trailing newline. An empty expression matches every line. Each input line that matches at least one of the patterns is written to the standard output.

`du`

The `du` utility displays the file system block usage for each file argument and for each directory in the file hierarchy rooted in each directory argument. If no file is specified, the block usage of the hierarchy rooted in the current directory is displayed.

1.7 Miscellaneous

`date`

When invoked without arguments, the `date` utility displays the current date and time. Otherwise, depending on the options specified, date will set the

date and time or print it in a user-defined way.

The `date` utility displays the date and time read from the kernel clock. When used to set the date and time, both the kernel clock and the hardware clock are updated.

1.8 Help!

`man <name>`

The `man` utility formats and displays the online manual pages. If you specify section, `man` only looks in that section of the manual. The name is normally the name of the manual page, which is typically the name of a command, function, or file.

The web

Lots of help is freely available on the web. Go ahead and Google it!

2

Getting started with python

"It is better to ask for forgiveness than permission."

– Proverb

2.1 Setting up python on your computer

There are many different installations of python out there and there are as many ways to go about setting up python on your python environement. We recommend that you download and install the free anaconda version of python v.2.7* as it will provide a comprehensive installation that will get you up and running quickly.

I will be using python 2.7, for various reasons. However, don't fret much about versions; once you have properly understood and learn to use one version of python, you can easily learn the differences and use the other. See more about python 2 vs python 3 online†

After python has been successfully installed you should be able to run python and ipython from the command line. Also available from the command line is a python development environment called *spyder*‡ that can be used to create and execute python code, and there is also a handy interface called *jupyter*§ or *ipython notebook*. But before we jump into things let us set up our python environment and workspace.

*https://docs.anaconda.com/anaconda/install/

†*e.g.*, https://tinyurl.com/yx2uuk3w

‡https://www.spyder-ide.org/

§https://jupyter.readthedocs.io/en/latest

2.2 Setting up your python path

With the anaconda installation of `python`, the easiest way to set up your `python` path is with the python path manager available through `spyder`.

Open spyder from the command line by typing:

```
$ spyder &
```

It may take a while for it to load the first time it is opened. Under the "python" pull down menu select "PYTHONPATH manager." You can add directories to the list with the "Add path" button. Directories that are on this list will be searched for modules and packages automatically and they will be searched in the order shown. So, any routines that you want access to without having to be in the proper present working directory should go in a directory on this list.

It would make sense to have all your code consolidated into one place on your hard drive that is easy to find and is organized in a logical fashion. Once you have all the desired directories added to this list, you will need to close and reopen `spyder` for the path to take effect. However, you may want to wait until you set up interactive graphing before you shut `spyder` down (next section).

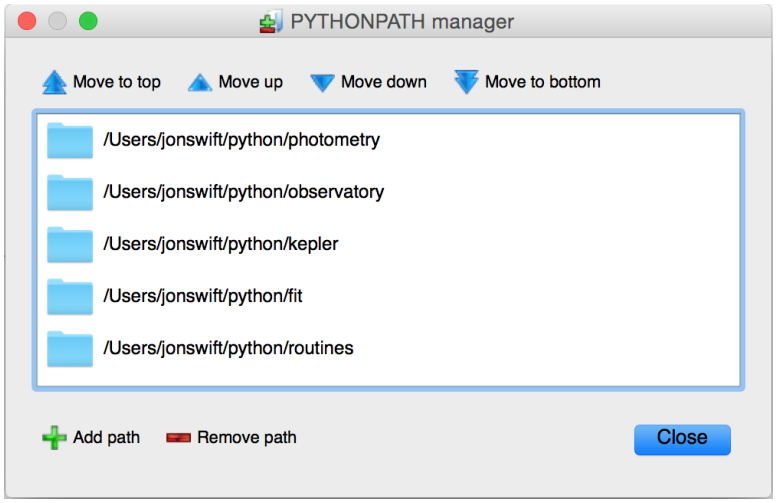

Figure 2.1: The PYTHONPATH manager window available from the "python" pull down menu in `spyder`.

Note that even though we are setting up our python path through `spyder`, this path will now be read for every python session you have, be it

through a python session initiated in a terminal window, jupyter, or command line.

2.3 Setting up interactive graphing

By default, `spyder` will produce plots "in-line" in your IPython console. This, in many or most circumstances is not ideal. Python has an interactive option that allows you to zoom in, pan, adjust subplot positioning, and save current view in a separate plot window. This feature can be turned on by first selecting "Preferences" under the "python" pull down menu. From the feature bar on the left side of the preferences window select "Ipython Console," then select the "Graphics" tab.

From the "Backend" pull down menu, select "Automatic." You must then click the "Activate" button in the bottom right of the window, and then "OK." To allow access to this new preference, you must quit `spyder` and reopen it.

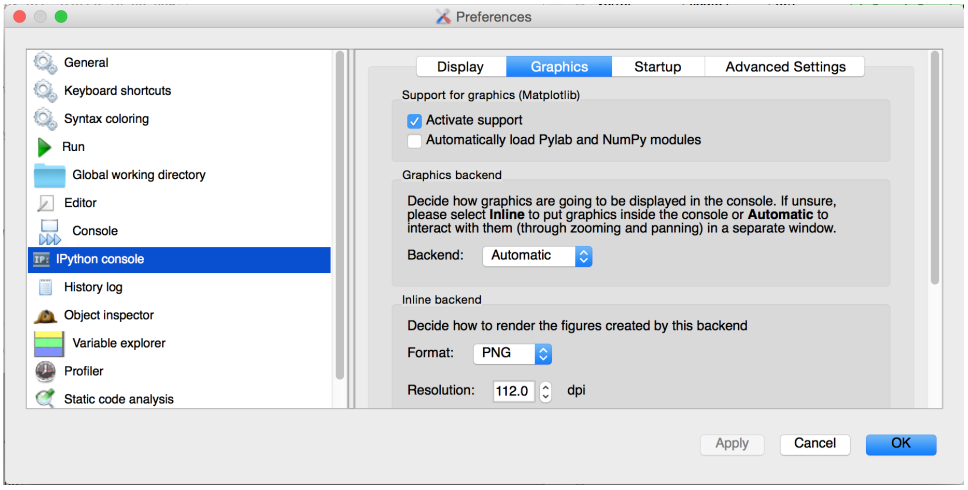

Figure 2.2: The preferences GUI in `spyder`. This figure accompanies the instructions on how to activate interactive plotting

2.4 The `spyder` Integrated Development Environment

We now take a bit of time to introduce the `spyder` graphical user interface (GUI). Anaconda provides shortcut icons to open spyder. But it is recommended that we start it from the command line. You can open up the python development environment, spyder, by typing:

```
$ spyder &
```

The ampersand will run this utility in the background (meaning that it will return a prompt so that further commands can be issued in your shell).

The spyder development environment is integrated, meaning it will run a python window as well as a text editor and other gadgets. The interface should look something like Figure 2.3. On the left side of the GUI there will be a text editor and the name of the file will be something like "untitled0.py"

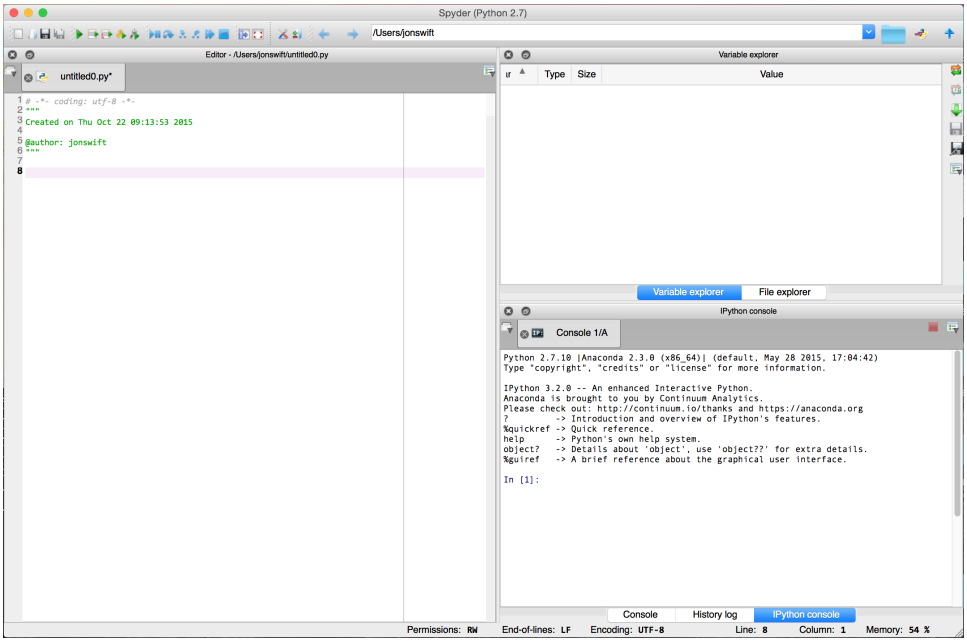

Figure 2.3: Graphical user interface for the `spyder` development environment. On the left is displayed the text editor. In the bottom right is the `ipython` console, and in the top right is the variable explorer. There are other display options available, though these are the default.

The bottom right of the `spyder` GUI is the IPython console. This is a direct interface to the python environment and where you can execute python commands line by line. As a first command, let us simply print hello. At the IPython prompt—which should be something like `In [1]:`—enter:
`print "Hello!"`
The console should return the string exactly as you typed it. If you received an error, that is a good sign that you are actually running python 3. In python 3, print statements must take a form that python 2 is also compatible with:
`print("I hope you have a great day.")`

Now let's say you want to make your greeting more personal. You can create a variable called `name` that is equal to your name: `name = "Jon"`
You should see the `name` variable in the upper right of the `spyder` GUI that will also give some other information about the variable. Now you can print a more personalized statement: `print "Hello, "+name+"!"`

If you want to make a function out of this little exercise, you can recall the lines you entered with the following command: `history`
This will list all the entries into your IPython console in your session (up to 200 lines, I believe). You can then copy and paste these lines from your console into your text editor for further processing.

Enter the lines of the following code listing, and save it as "hello.py" in our working directory. Choose "Save" from the "File" pull down menu. It will open a GUI prompt for the file name and the directory in which you would like to save it. If there is confusion about the path to your working directory, you may type `pwd` in the IPython console.

Listing 2.1: `hello.py`—Getting acquainted with python.

```
1 def greeting(name):
2     print "Hello, "+name+"!"
3     return
```

Once this code is saved, you can run it by either typing:
`%run hello`
in the IPython console, or by pressing green "Play" arrow in the `spyder` menu bar at the top of the GUI. Once this is done you can type:
`greeting("Jon")`
in the IPython console to execute it.

There is much more functionality to the `spyder` IDE that we will not go

into here, but you can dig as deep as you wish online[¶]

2.5 `jupyter` (a.k.a. the ipython notebook)

The ipython notebook, now called the jupyter notebook (pronounced like the planet, despite the weird spelling), is a powerful coding interface particularly for learning python. It is built around an HTML interface and runs in your browser.

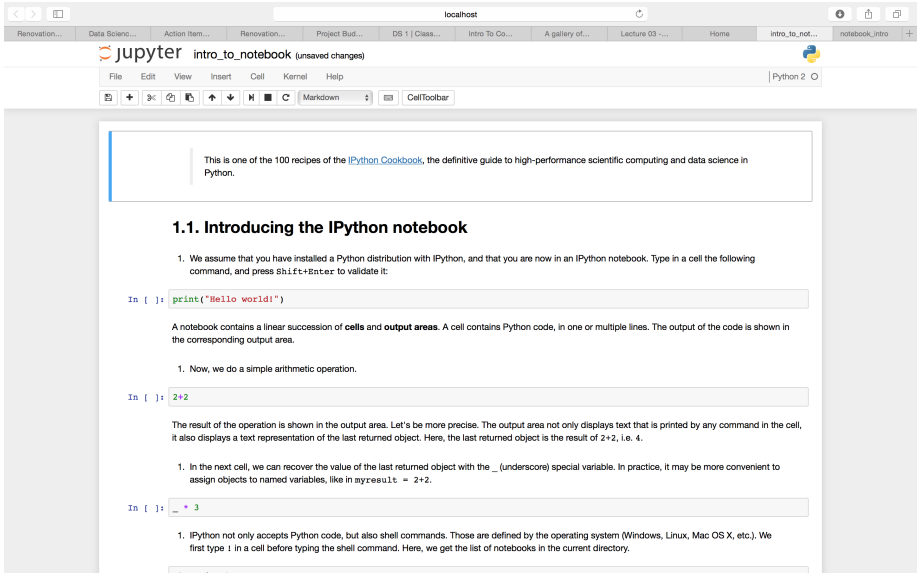

Figure 2.4: Example of an `jupyter notebook` shown through a browser interface. Markdown cells with standard text as well as code cells that can be executed are shown.

In addition to running your code, `jupyter` stores code and output, together with markdown notes, in an editable document called a notebook. When you save it, this is sent from your browser to the notebook server, which saves it on disk as a JSON file with a `.ipynb` extension.

The notebook server, not the kernel, is responsible for saving and loading notebooks, so you can edit notebooks even if you don't have the kernel for that language—you just won't be able to run code. The kernel doesn't

[¶] *e.g.*, https://pythonhosted.org/spyder

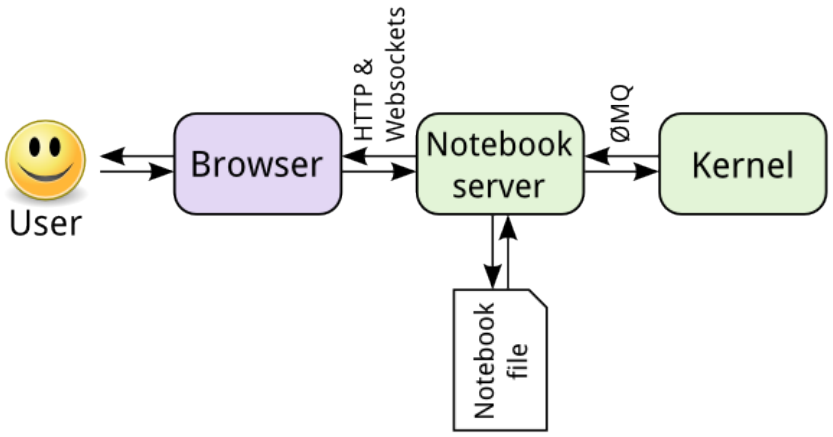

Figure 2.5: Flow chart of how jupyter works (from jupyter manual webpage).

know anything about the notebook document: it just gets sent cells of code to execute when the user runs them.

To start a `jupyter notebook` session from the command line enter:

```
$ jupyter notebook
```

This will automatically start (or open a new tab in) your default browser with a home screen showing the contents of your current working directory. You can navigate through folders and you can start a `.ipynb` file by double clicking. This will open a new browser tab with the contents of the python notebook. It will also start a python kernel so that code can be executed.

The notebook can be edited and saved or copied. To close a notebook, you can close the tab or choose "Close and Halt" from the "File" pull down menu. From the "Home" page, the kernel that was running the notebook can be killed by selecting that file and clicking the "Shutdown" button.

The details of how to navigate, save, and edit jupyter notebooks are best done (of course) through a tutorial jupyter notebook. So, we will leave that as an exercise to the reader. There are many examples online[‖]

[‖] *e.g.*, https://jupyter.readthedocs.io/en/latest/tryjupyter.html

3

Version Control with `git`

By far, the most widely used modern version control system in the world today is `git`. `git` is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel. A staggering number of software projects rely on `git` for version control, including commercial projects as well as open source. Developers who have worked with `git` are well represented in the pool of available software development talent and it works well on a wide range of operating systems and IDEs (Integrated Development Environments).

Having a distributed architecture, `git` is an example of a DVCS (Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in `git`, every developer's working copy of the code is also a repository that can contain the full history of all changes.

In addition to being distributed, `git` has been designed with performance, security and flexibility in mind. A visual overview of what is happening with `git` is shown in Figure 3.1.

We will not be using `git` to its full potential. So this section is divided up into the most important functionality that we will need. I hope that it will serve as a working introduction into the world of `git`.

3.1 Installing `git`

For a modern Mac, `git` comes standard with XCode command line tools. You can install these through the App Store. It is also a standard application for Linux users. Windows users will have a much more difficult time getting

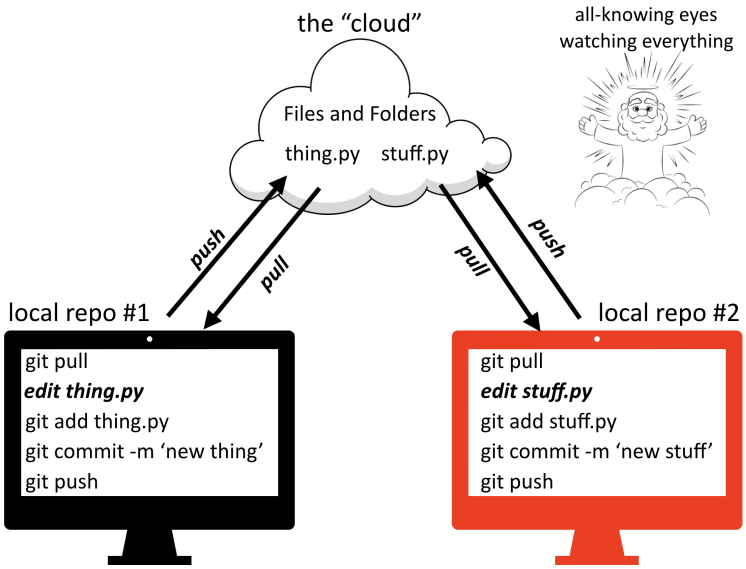

Figure 3.1: Conceptual drawing of what is happening when using the `git` with remote tracking.

git to work from the command line. Often it is a better option to use a third party software on Windows to manage your `git` repositories, like GitKraken*

3.2 Getting started

Input your name into your configuration file so that `git` can track the changes you made.
`git config --global user.name "Your Name"`
You can input your email so that changes to the code, or other relevant updates can be sent to you via email:
`git config --global user.email yourhandle@example.com`
When you commit `git` will ask you to input a comment. It will open a text editor that you specify with the following command:
`git config --global core.editor open (or emacs, or pico, or whatever)`
List your configuration settings with: `git config --list`

*https://www.gitkraken.com/

3.3 Getting help

Three ways to get help directly from the command line:
`git help <something>`
`git <something> --help`
`man git-<something>`
Online there are also ample resources, for example: `http://git-scm.com/doc`

3.4 Setting up repositories

To set up `git` tracking in an existing directory, you need to navigate to that directory and then type:
`git init`
You can then add existing files to the tracking system by adding them:
`git add *.c`
`git add LICENSE`
And then committing the changes that you have made:
`git commit -m "initial project version"`
If you do not type in the message at the end the default editor will open and require that you type a message.
If you'd like to clone an existing project using `https` protocol:
`git clone https://github.com/libgit2/libgit2 mylibgit <local-name>`

3.5 Recording changes to the repository

Check the status of your local repository by typing:
`git status`
This will show you which files are being tracked, which are staged, which are committed, and which have changes that are not staged. For more detail about which changes have been made to files but not committed, type:
`git diff`
To add additional files to be tracked:
`git add <filename>`
After changes have been made you must stage these files for a commit with the add command. Once you have made sufficiently stable changes to your files and staged them, you commit these changes with:
`git commit -m "comment goes here"`
If you do not type a comment, then the default editor will be opened and the commit will not proceed until a comment is written. You can also bypass

the staging with a `-a` flag.

To remove a file, you would type:
`git rm <filename>`
This stages the file for removal such that on the next commit the file will be removed and no longer tracked.

To view the commit history you would us the log command in `git` with its various flags:
`git log`
You can check out:
`http://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History`
for all the gory details.

3.6 Undoing things

A common error is to commit something before your were really ready to. There is a command that allows the ammendment of a commit:
`git commit --amend`
To unstage a staged file you can use the reset command:
`git reset HEAD <filename>`
This is best checked by typing `git status` beforehand as the default messaging typically gives the command for unstaging staged files.
To unmodify a modified file you can type:
`git checkout -- <filename>`
Again, `git status` will be default give you the explicit command.

It's important to understand that `git checkout -- <filename>` is a **dangerous** command. Any changes you made to that file are gone—you just copied another file over it. Don't ever use this command unless you absolutely know that you don't want the file.

If you have made changes to the repo and none of them were useful, you may want a hard reset of the repo. You can execute that with the following:
`git log`
From this you will get your last push commit hash key
`git reset --hard <your commit hash key>`
You can also use the stash command which squirrels away your changes and keeps them accessible (though not straightforward to get back):

`git stash save --keep-index`
After that, you can drop that stash by issuing the command:
`git stash drop`.
If it is a single file that you want to revert:
`git checkout path/to/file/to/revert`
And to revert all unstaged files:
`git checkout -- .`
Be sure to include the dot at the end.

3.7 Working with remotes

To see which remote servers you have configured, you can run:
`git remote -v`
If you've cloned your repository, you should at least see origin—that is the default name `git` gives to the server you cloned. To add a remote repository:
`git remote add <shortname> <url>`

To get data from your remote projects, you can run:
`git fetch <shortname>`
It's important to note that this command pulls the data to your local repository—it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready. Alternatively, you can run:
`git pull`
This will automatically fetch and then merge any updates pushed to the remote repository with your local repository.

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple:
`git push <remote-name> <branch-name>`
You can inspect a particular remote with:
`git remote show <remote-name>`
If you want to rename the shortname of a remote you can type:
`git remote rename <shortname> <newshortname>`

3.8 Branches in a nutshell

To view all accessible branches in a repository:
`git branch`
To create a new branch and switch to that branch:
`git checkout -b <branchname>`
You can now edit files, add and commit them. You cannot change to another branch until your changes are committed (there are ways to get around this (namely, stashing and commit amending).

After you have made sufficient changes to your branch and you want to merge them to the master. You must stage and commit the changes in your branch, then switch back to the master branch with:
`git checkout master`
Followed by the merge command:
`git merge <branchname>`

To delete a branch named test:
`git branch -d test`
If there are unmerged changes it will complain to you. However, if you want to destroy the branch anyway you can type:
`git branch -D test`

3.9 Setting up a remote branch with tracking

To set up a remote branch with tracking you will need to issue the checkout command with the `-b` feature enabled.
`git checkout -b feature-branch-name`
From here you can edit files, add files to your commit queue, and then commit them according to the procedures outlined above. Then you can push your branch upstream with the following command.
`git push -u origin feature-branch-name`

3.10 Adding a remote branch locally

If you would like to add a remote branch to your local repository, you can do so with the following commands
`git fetch`

```
git checkout branch-name
```

Here `branch-name` is the name of the remote branch.