

Merely Useful: Novice Python

Madeleine Bonsma-Fisher, Kate Hertweck, Damien Irving, Luke Johnston, Christina Koch

Contents

Chapter 1	Overview	11
1.1	Why isn't all of this normal already?.....	11
1.2	Acknowledgments	11
Chapter 2	Novice Goals	13
2.1	Personas	13
2.2	Getting started.....	15
2.3	Data manipulation	16
2.4	Plotting	17
2.5	Development	17
2.6	Data analysis.....	18
2.7	Version control	19
2.8	Publishing	19
2.9	Reproducibility	20
2.10	Collaboration	21
Chapter 3	Introduction	23
3.1	Who are these lessons for?	23
3.2	What does “done” look like?.....	23
3.3	What will we use as running examples?.....	23
Chapter 4	Getting Started with Python	25
4.1	Questions	25
4.2	Introduction to Spyder.....	25

4.3	Running Code in the Console	25
4.4	Running Code via Scripts	27
4.5	Variables, Objects, Methods	29
4.6	Getting Help in Spyder	31
4.7	Getting Help Online.....	31
4.8	Exercises	32
4.9	Key Points	33
Chapter 5	Development	35
5.1	Questions	35
5.2	Functions.....	35
5.3	How to make programs indicate that something has gone wrong?	41
5.4	Packages.....	44
5.5	How to get help online	50
5.6	Key Points	50
5.7	Exercises	51
Chapter 6	Publishing.....	53
6.1	Questions	53
6.2	Why should I share my work on the internet?....	53
6.3	What does it take to get a webpage online?	53
6.4	How do I get my work on the web?	54
6.5	How do I link to other pages, files or images?.....	60
6.6	Exercise: Add a website to an existing project ...	64
Chapter 7	Version Control.....	65
7.1	Questions	65
7.2	What is Version Control?.....	65

Contents	5
7.3 Creating a Repository	66
7.4 Tracking Changes	67
7.5 Examining Previous Versions	71
7.6 Online Repositories	73
7.7 Conflicts	76
7.8 Using Local vs Remote Repositories	78
7.9 To Git-finity, and Beyond	78
7.10 Exercises	79
7.11 Key Points	80
Appendix A License	81
Appendix B Code of Conduct	83
B.1 Our Standards	83
B.2 Our Responsibilities	83
B.3 Scope	84
B.4 Enforcement	84
B.5 Attribution	84
Appendix C Contributing	85
C.1 Style Guide	85
C.2 Setting Up	86
Appendix D Glossary	87
Appendix E Novice Python Learning Objectives	99
Appendix F Novice Python Key Points	101

List of Tables

List of Figures

4.1	Spyder	26
4.2	A Short Script	28
5.1	Rich rendering of the docstring in Spyder	42
7.1	Getting Started	67
7.2	Important Panes.....	68
7.3	commit-cycle.....	69
7.4	revert-image.....	72
7.5	remote-image	74
7.6	Committing	76
7.7	The Commit Process	78

1 Overview

It's still magic even if you know how it's done.
— Terry Pratchett

FIXME: general introduction

- Be able to do the steps in Yenni et al. (2019).

1.1 WHY ISN'T ALL OF THIS NORMAL ALREADY?

Nobody argues that research should be irreproducible or unsustainable, but “not against it” and actively supporting it are very different things. Academia doesn't yet know how to reward people for writing useful software, so while you may be thanked, the extra effort you put in may not translate into job security or decent pay.

And some people still argue against openness, Being open is a big step toward a (non-academic) career path, which is where approximately 80% of PhDs go, and for those staying in academia, open work is cited more often than closed (FIXME: citation). However, some people still worry that if they make their data and code generally available, someone else will use it and publish a result they have come up with themselves. This is almost unheard of in practice, but that doesn't stop people using it as a boogeyman.

Other people are afraid of looking foolish or incompetent by sharing code that might contain bugs. This isn't just impostor syndrome: members of marginalized groups are frequently judged more harshly than others (FIXME: citation).

1.2 ACKNOWLEDGMENTS

This book owes its existence to the hundreds of researchers we met through Software Carpentry and Data Carpentry. We are also grateful to Insight Data Science for sponsoring the early stages of this work, to everyone who has contributed, and to:

- *Practical Computing for Biologists* Haddock and Dunn (2010)
- *Effective Computation in Physics* Scopatz and Huff (2015)
- “A Quick Guide to Organizing Computational Biology Projects” Noble (2009)
- “Ten Simple Rules for Making Research Software More Robust” Taschuk and Wilson (2017)
- “Best Practices for Scientific Computing” Wilson et al. (2014)
- “Good Enough Practices in Scientific Computing” Wilson et al. (2017)

2 Novice Goals

This outline describes the questions that the novice courses on R and Python will answer. The advanced course can then assume that learners have hands-on experience with these topics but nothing more.

2.1 PERSONAS

2.1.1 ANYA

Anya is a professor of neuropsychology who is responsible for teaching her department's introduction to statistics to 1100 first-year students every year. (Students complain that the Stats department's introductory course is too theoretical and requires more programming knowledge than they have.) When she finds time for it, her research focuses on color perception in infants.

Over the past nine years, Anya has designed and run a dozen experiments on 50-100 infant subjects each and analyzed the results using SPSS and more recently R (which she taught herself during a sabbatical). She has never taken a programming course, and suffers from impostor syndrome when talking to colleagues who are using things like GitHub and R Markdown.

Anya would like to figure out how to use R to teach her intro stats course, which currently uses a mixture of Excel and SPSS. She would like to learn more about time series analysis to support her research, and about tools like Git and R Markdown.

This guide has modular lessons and exercises that she can adapt to use in her course, and suggestions for how to make learning interactive with a large class size. She also finds helpful instructions for applying time series analysis to data using R.

2.1.2 EXTON

Exton taught business at a community college before joining a friend's startup, and now does community management for a company that

builds healthcare software. He still teaches Marketing 101 every year to help people with backgrounds like his.

Exton uses Excel to keep track of who is registered for webinars, workshops, and training sessions. Some of these spreadsheets are created from CSV files produced by a web-scraping script a summer intern wrote for him a couple of years ago. Exton doesn't think of himself as a programmer, but spends hours creating complicated lookup tables in multi-sheet spreadsheets to help him figure out how many webinar attendees turn into community contributors, who answers forum posts most frequently, and so on.

Exton knows there are better ways to do what he's doing, but feels overwhelmed by the flood of blog posts, tweets, and "helpful" recommendations he receives from members of the company's engineering team. He wants someone to tell him where he should start and how long it will take whatever he learns to pay off.

Exton finds 'Merely Useful' after some Googling, and sees an example of data analysis with spreadsheet data that looks really similar to what he's trying to do. He carefully works through that particular example, then goes back and works through some of the earlier material in the book. He can tell that it won't take long to get this to work with his data.

2.1.3 IRWIN

Irwin, 18, is five months into an undergraduate degree in urban planning. He's read lots of gushing articles in *Wired* about data science, and was excited by the prospect of learning how to do it, but dropped his CS 101 course after six weeks because nothing made sense. (His university's computer science department uses Haskell as an introductory programming language...) He is doing better in Anya's course (which he is taking as an elective) but still spends most of his time copying, pasting, and swearing.

Irwin did well in his high school math classes, and built himself a home page with HTML and CSS in a weekend workshop in grade 11. He knows how to do simple calculations in Excel, has accounts on nine different social media sites, and attends all of his morning classes online.

Anya mentions this guide in one of her classes, and Irwin downloads the PDF to read on the bus. He loves the examples that use urban data, and right away he has tons of ideas about where to get more cool data

to analyze. His urban data science blog is already taking shape in his head.

2.1.4 CAMILLA

Camilla recently started a job as an assistant professor. Her department (Medieval Studies) is trying to develop a digital humanities data-science-heavy undergraduate program, and the undergraduate chair thinks that Camilla has the most programming experience in the department and has asked her to develop an introduction to programming course for humanities students.

Camilla has dabbled in natural language processing and has learned Python over the course of her previous work, but she has no experience teaching programming and she's not sure what the best way is to teach beginners. She doesn't want to start from scratch to create a course out of nothing. She also isn't sure which programming language the new program should focus on.

She finds 'Merely Useful' and feels relieved: she can pretty much use the book as-is for her course. She looks up the examples of text and image analysis and compares how both R and Python approach those kinds of data to help her make a decision about which language to teach.

2.1.5 JORDAN

Jordan is a third-year undergraduate student in ecology. Two months ago she started working part-time for a professor in her department, and she's beginning to collect and analyze data from her own experiments with fruit flies. Her professor has asked her to learn R to do her analysis and suggested that she sign up for the introduction to quantitative data analysis in R course that the ecology department offers. The course is just starting, and it uses 'Merely Useful' as the textbook.

Jordan can't wait to apply her new programming knowledge to her data, so she starts reading ahead and trying to use her own data in some of the book's examples. As she works through examples, she realizes that she'll need to change a few things about how she records her data in spreadsheets so that it will be easier to analyze in R.

2.2 GETTING STARTED

- What are the different ways I can interact with software?

- console
- scripts
- How can I find and view help?
 - In the IDE
 - Stack Overflow
- How can I inspect data while I'm working on it?
 - table viewers

2.3 DATA MANIPULATION

- How can I read tabular data into a program?
 - what CSV is, where it comes from, and why people use it
 - reading files
- How can I select subsets of my data?
 - select
 - filter
 - arrange
 - Boolean conditions
- How can I calculate new values?
 - mutate
 - ifelse
- How can I tell what's gone wrong in my programs?
 - reading error messages
 - the difference between syntax and runtime errors
- How can I operate on subsets of my data?
 - group
 - summarize
 - split-apply-combine
- How can I work with two or more datasets?
 - join
- How can I save my results?
 - writing files
- What *isn't* included?
 - anything other than reasonably tidy tabular data
 - map
 - loops and conditionals

2.4 PLOTTING

- Why plot?
 - summary statistics can mislead
 - Anscombe's Quartet and the DataSaurus dozen
- What are the core elements of every plot?
 - data
 - geometric objects
 - aesthetic mapping
- How can I create different kinds of plots?
 - scatter plot
 - line plot
 - histogram
 - bar plot
 - which to use when
- How can I plot multiple datasets at once?
 - grouping
 - faceting
- How can I make misleading plots?
 - showing a single central tendency data point instead of the individual observations
 - saturated plots instead of for example violins or 2D histograms
 - picking unreasonable axes limits to intentionally misrepresent the underlying data
 - not using perceptually uniform colormaps to indicate quantities
 - not thinking about color blindness
- What *isn't* included?
 - outliers
 - interactive plots
 - maps
 - 3D visualization

2.5 DEVELOPMENT

- How can I make my own functions?
 - declaring functions
 - declaring parameters
 - default values

- common conventions
- How can I make my programs tell me that something has gone wrong?
 - validation (did we build the right thing) vs. verification (did we build the thing correctly)
 - assertions for sanity checks
- How can I ask for help?
 - creating a reproducible example (reprex)
- How do I install software?
 - what *is* a package?
 - package manager
- What *isn't* included?
 - code browsers, multiple cursors, and other fancy IDE tricks
 - virtual environments
 - debuggers

2.6 DATA ANALYSIS

- How can I represent and manage missing values?
 - NA
- How can I get a feel for my data?
 - summary statistics
- How can I create a simple model of my data?
 - formulas
 - linear regression
 - adding a best fit straight line on a scatterplot
 - understanding what the error bands on a “best fit” straight line mean
 - k-means cluster analysis
 - frame these as exploratory tools for revealing structure in the data, rather than modelling or inferential tools
- How can I put people at risk?
 - algorithmic bias
 - de-anonymization
- What *isn't* included?
 - statistical tests
 - multiple linear regression
 - anything with “machine learning” in its name

2.7 VERSION CONTROL

- What is a version control system?
 - a smarter kind of backup
- What goes where and why?
 - local vs. remote storage (physically)
 - local vs. remote storage (ethical/privacy issues)
- How do I track my work locally?
 - diff
 - add
 - commit
 - log
- How do I view or recover an old version of a file?
 - diff
 - checkout
- How do I save work remotely?
 - push and pull
- How do I manage conflicts?
 - merge
- What *isn't* included?
 - forking
 - branching
 - pull requests
 - git reflow –substantive –single-afferent-cycle –ia-ia-rebase-fhtagn ...

2.8 PUBLISHING

- How do static websites work?
 - URLs
 - servers
 - request/response cycle
 - pages
- How do I create a simple HTML page?
 - head/body
 - basic elements
 - images
 - links
 - relative vs. absolute paths

- How can I create a simple website?
 - GitHub pages
- How can I give pages a standard appearance?
 - layouts
- How can I avoid writing all those tags?
 - Markdown
- How can I share values between pages?
 - flat per-site and per-page configuration
 - variable expansion
- What *isn't* included?
 - templating
 - filters
 - inclusions

2.9 REPRODUCIBILITY

- How can I make programs easy to read?
 - coding style
 - linters
 - documentation
- How can I make programs easy to re-use?
 - Taschuk's Rules
- How can I combine explanations, code, and results?
 - notebooks
- Where does stuff actually live on my computer?
 - directory structure on Windows and Unix
 - absolute vs. relative paths
 - significance of the working directory
 - data on disk vs. data in memory
- How should I organize my projects?
 - Noble's Rules
 - RStudio projects
- How should I keep track of my data?
 - simple manifests
- What *isn't* included?
 - build tools (Make and its kin)
 - continuous integration
 - documentation generators

2.10 COLLABORATION

- What kinds of licenses are there?
 - open vs. closed
 - copyright
- Who gets to decide what license to use?
 - it depends...
- What license should I use for my publications?
 - CC-something
- What license should I use for my software?
 - MIT/BSD vs. GPL
- What license should I use for my data?
 - CC0
- How should I identify myself and my work?
 - DOIs
 - ORCIDs
- How do I credit someone else's code?
 - citing packages, citing something from GitHub, giving credit for someone's answer on StackOverflow...
- What's the difference between open and welcoming?
 - evidence for systematic exclusion
 - mechanics of exclusion
- How can I help create a level playing field?
 - what's wrong with deficit models
 - allyship, advocacy, and sponsorship
 - Code of Conduct (remove negatives)
 - curb cuts (adding positives for some people helps everyone else too)
- What *isn't* included?
 - how to run a meeting
 - community management
 - mental health
 - assessment of this course

3 Introduction

FIXME: general introduction.

3.1 WHO ARE THESE LESSONS FOR?

FIXME: personas

3.1.1 SUMMARY

FIXME: elevator pitch

3.1.2 PREREQUISITES

FIXME: prerequisites

3.2 WHAT DOES “DONE” LOOK LIKE?

FIXME: end goal

3.3 WHAT WILL WE USE AS RUNNING EXAMPLES?

FIXME: introduce running examples

4 Getting Started with Python

4.1 QUESTIONS

- How do I run Python code?
- What are some basic actions I can do with Python?
- How do I get help?

As stated in the intro, our overall goal is to work with people, programs and data. In this section, we will focus on programs and data as we learn how to run Python code and how data is stored and accessed on a computer.

4.2 INTRODUCTION TO SPYDER

Starting with programs - over the entirety of this book, we'll be writing programs (or, in verb form, *programming*) in order to accomplish our goals of working with data on the computer. Programming is one way to make a computer do something for us. Instead of clicking, we'll mostly be typing; instead of doing what someone else has pre-defined, we'll have a lot of flexibility to do what we want.

Just like using a web browser to access websites, and a program like Word to write documents, it's helpful to have a program on your computer that is designed to make it easy to write and run code. This kind of program is called an "IDE" or Integrated Development Environment.

The one we'll be using in this book for writing Python code is called Spyder. Spyder (like many IDEs) opens up and has many panes, that each have a different purpose (Figure 4.2).

We'll take it slowly and introduce the purpose of each pane one-by-one. To start with, we'll focus on the left and lower-right panes, as that's where the bulk of our coding will happen.

4.3 RUNNING CODE IN THE CONSOLE

Now that we have our code-writing-and-running environment open, it's time to actually run some code.

FIGURE 4.1 Spyder

In the bottom right pane, you can see something called the “console”. This is a program that is constantly ready and waiting to take in code and run it; as a Python console, the language it expects to see is Python. Here’s an example you can type or copy in:

```
print("It was the best of times, it was the worst of times.")
```

```
## It was the best of times, it was the worst of times.
```

As you might guess from the name, the Python `print()` function takes in text (indicated by the quotes) and then prints the text back out to the console. Python has many built-in functions, that allow you to do different things:

```
len("Sydney Carton")
```

```
## 13
```

```
sorted(["Samuel", "Nathaniel", "Augustus", "Tracy"])
```

```
## ['Augustus', 'Nathaniel', 'Samuel', 'Tracy']
```

```
min([4, 2/3, -3, 6.25])
```

```
## -3
```

```
pow(5,3)
```

```
## 125
```

Part of learning to program is learning some of the core Python functions and what they do; in upcoming chapters, we'll focus on functions that allow you to read in and manipulate data. At some point it may be helpful to write your *own* functions, which we'll also cover in another chapter.

4.4 RUNNING CODE VIA SCRIPTS

Can you imagine a situation where continuing to type code into the console could become tedious or challenging?

Here are some examples:

- Wanting to run the same piece of code many times (you *can* use the up-arrow to see previous commands - try it! - but if you run many commands, your history will quickly become more cluttered.
- Wanting to organize long sections of code in a meaningful way and work on subsections separately.
- Wanting to save the code as a text file that can be stored on the computer's hard drive and shared with collaborators.

So we're motivated to find a way to write and run code where we're keeping a record as we go along, and it is easy to run code again, in chunks.

The answer to these challenges is to move to the left pane of the Spyder IDE. This is a script: a text file with Python code. Try copying in some of the commands we've already run (Figure 4.2):

```
print("It was the best of times, it was the worst of times.")

## It was the best of times, it was the worst of times.

sorted(["Samuel", "Nathaniel", "Augustus", "Tracy"])

## ['Augustus', 'Nathaniel', 'Samuel', 'Tracy']
```

FIGURE 4.2 A Short Script

Now these commands are written in the script, but nothing has happened yet. In order to run the commands from the script (just like we did in the console), we'll have to use various run commands. You can see these under the “Run” menu; but typically you'll want to learn at least one or two keyboard shortcuts so it becomes quick and easy to run code that's written in a script. Try “running” the lines in your script. You should see their output in the console on the bottom right.

4.4.1 ORGANIZING SECTIONS WITH CODE CELLS

When writing code it is helpful to have granular control over what code is executed when and we often don't want to run the entire script. A helpful feature of scripts are code cells, which are separated by `#%%`. If we would rewrite the code above so that each line is in a different code cell, it would look like this.

```
###  
print("It was the best of times, it was the worst of times.")  
###
```

```
## It was the best of times, it was the worst of times.
```

```
sorted(["Samuel", "Nathaniel", "Augustus", "Tracy"])
```

```
## ['Augustus', 'Nathaniel', 'Samuel', 'Tracy']
```

To run a code cell, we can press **Ctrl+Enter**. To run a code cell and advance to the next, we can press **Shift+Enter**. Code cells facilitate developing and troubleshooting sections of the code one at a time. A common scenario where this is useful in data science is when you read in data from a file in one code cell and start working and iterating on the analysis part in the next code cell without the need to reread the data every time you want to test your new analysis. We will talk more about code cell workflows later.

Sometimes, when you want to experiment or check something, it can make sense to write and run Python code in the console. However, most of the time, you'll want to write and run your code from a script. Using scripts has the benefit of saving your work, commonly as a text file with the `.py` extension, while also being able to run the code just like in the console. The script then has the added benefit of being easy to share and runnable outside of the IDE.

We have now told the computer what to do by using Python code, and we have run that code in two different ways within our “workbench” - the Spyder IDE. Let's see what else we can use in this environment to help us.

4.5 VARIABLES, OBJECTS, METHODS

Besides running functions that *do* something (as above), we'll also want to use Python to keep track of information that we're using throughout our analysis. We save that information by giving it a name (called a variable), using the name, an equals sign (=) and then the information to save. For example:

```
message = "It is a far, far better thing that I do, than I have ever done"
name_length = len("Sydney Carton")
```

The actual values being referenced by the variable (to the right of the equals sign) are stored by the programming language as an object. There are different types of objects based on the type of information you're using. On the top right pane, there's a tab that says "Variable Explorer". If you click on that, you'll see the variables and corresponding objects you just created, with some additional information about the *Type* - `str` (string of letters), `int` (integer), `float` (a "floating point" number, or decimal) and more.

Objects can also be referenced later in functions by using their variable name:

```
print("My message:", message)
```

```
## My message: It is a far, far better thing that I do, than I have ever done
```

```
print("Letters in name:", name_length)
```

```
## Letters in name: 13
```

Most types have special functions specific to that type, called [methods][method]. Methods look like functions (`name()`), but instead of putting the input information inside the parentheses (as we did with the functions above), methods are tacked onto the end of a variable, assuming that the "input" information the method uses is coming from that variable.

For example, what happens if we use the following string methods:

```
message.upper()
```

```
## 'IT IS A FAR, FAR BETTER THING THAT I DO, THAN I HAVE EVER DONE'
```

```
message.split()
```

```
## ['It', 'is', 'a', 'far,', 'far', 'better', 'thing', 'that', 'I', 'do,', 'than', 'I',
```

Here, it's assumed that the contents of the `message` variable are the information used by the `upper()` and `split()` string methods. Note how an `upper()` method makes sense for strings, but wouldn't make sense for a numeric object.

4.6 GETTING HELP IN SPYDER

That top-right pane can give us additional information; one of the other tabs says "Help". The main way to look up a particular function or type is to type it into the Console and then type the Help shortcut (which is `CTRL-I`). So, if you wanted to look up information about the `len()` function, we used above, you can type the following into the console and then use the help shortcut:

```
len
```

Typing the full name of the function (without pressing `ENTER`) with the Help tab selected will also trigger the help page about that function:

```
len()
```

You can see what methods are available for an object by typing the variable name into the console or script pane and then pressing `TAB`. This will bring up a list of possible methods to use.

4.7 GETTING HELP ONLINE

Sometimes, however, your questions won't fit well within the built-in help pages, available through Spyder. This often happens when you need to do something completely new and you don't know where to start. In that scenario, there are multiple internet resources that are helpful:

- blogs
- question pages on Stack Overflow
- Twitter

Sometimes these resources can be overwhelming and confusing. Some things to look for on a Stack Overflow answer:

- Is the question clearly stated and is there example code? If you're debugging, does the example code look like yours?
- Answers have upvotes and downvotes. Is there one clear answer that has a lot of upvotes in response to a question?
- How complex is the answer? While some questions will necessarily have a complicated answer, for many common programming tasks, there should be a few-line solution to the question.
- Do you recognize any terms in the solution? If you don't, are there other terms that you could search on?

Conventions

One tool in using help pages (and in reading the rest of the book) will be understanding the conventions that writers use when describing many of the terms introduced in this chapter. For example

`name()`

indicates that something is a function, where:

`type.name()`

would indicate something is a method. Both have parentheses to indicate that they are actions with (potentially) inputs and outputs.

Some other conventions used in this book are: `* folder_name/` for a folder `* variable_name` for variables `* column_name` for columns

4.8 EXERCISES

1. Run each of the functions above. Can you explain what each function expects as input, and what kind of output it produces?
2. What would be the pros and cons of using the console versus a script in each situation?
 - Writing a data analysis with multiple steps.
 - Opening a new data set and exploring its dimensions.
 - Checking the value of a variable.
3. You've saved a string of interesting data to a variable.

```
sentence = "One fish, two fish, red fish, blue fish."
```

How can you count the occurrences of the letter "f"?

4. After googling "how to read csv file python", these two answers pop up: <https://stackoverflow.com/questions/41585078/how-do-i-read-and-write-csv-files-with-python/41585079>
<https://stackoverflow.com/questions/5788521/reading-a-csv-file-using-python>

- What is useful about each page?
 - Do you understand all of the code in the second page?
 - There are many options given in the first Stack Overflow page. Which one seems like a good place to start?
 - Both pages include python libraries (extra tools), which are added to the code using `import`. Based on this information, looking at the options shown in these two pages, what are two libraries that can be used to read in csv files?
5. Try reading a csv file:

```
import pandas
data = pandas.read_csv("measurements.csv")
```

Which pane in Spyder will show our new `data` object? What happens when you click on it?

4.9 KEY POINTS

- Spyder is an Integrated Development Environment (IDE) for writing Python code.
- You can run Python via a script or console; for most scenarios, we write and execute code from a script.
- IDEs like Spyder have shortcuts and help pages to facilitate writing code.
- Actions in Python are performed by functions and methods.
- Information in Python is stored as objects that are labeled with variables.

5 Development

5.1 QUESTIONS

- What are functions and how can I make my own?
- How can I make my programs tell me that something has gone wrong?
- How can I ask for help online?
- What are packages and how do I install them?

5.2 FUNCTIONS

Functions are like recipes. You give a few ingredients as input to a function, and it will generate an output based on these ingredients. Just as when following a recipe, both the ingredients and the instructions will influence the final result.

In Python, the inputs to a function are not called ingredients, but rather arguments, and the output is referred to as the return value of the function. A function does not technically need to return a value, but often does so. Functions facilitate reusing chunks of code in a way that is more readable and reproducible than cutting and pasting several lines of code. E.g. if our data analysis code is broken down into functions, we could readily use it with many different data sets by changing the input data path, but leaving the rest of the code the same.

Well chosen function names also clarifies the flow of analysis. For example, imagine that you open a file with the following lines of code within it.

```
images = read_in_images(file_paths)
gray_images = convert_to_grayscale(images)
brightest_image = find_brightest_image(gray_images)
```

Just by looking at the function names, it is clear what this code is intended for and its main flow of operations is immediately visible. Inside each of these functions there might be 10-20 lines of code, so if we would not have modularized the code into separate functions with well chosen

names, it would take longer to understand its overall purpose since there would be 30-60 lines of code to read instead of just three.

5.2.1 CREATING FUNCTIONS

There are many useful functions already built into Python, and the ability to create your own allows you to string together any sequence of operations in ways that are tailored to your workflow. The `def` keyword lets us define a function with a name of our choice and an arbitrary number of input parameters.

```
def sum_two_numbers(num1, num2):  
    return num1 + num2
```

This function accepts two input parameters, `num1` and `num2`, and returns their sum. Just as with variable names, function names are preferably written in `snake_case` (see the style guide for details), and avoid existing Python keywords and built-in names (a list of these is available here, but instead of memorizing that list you can type the desired name into the Python interpreter to find out if it already exists).

To execute the operations listed in the function, we can call the function and pass the two numbers we want to add as the arguments to the function.

```
sum_two_numbers(2, 5)
```

```
## 7
```

The returned value can be assigned to a variable:

```
number_sum = sum_two_numbers(2, 5)  
number_sum
```

```
## 7
```

A more versatile function could add any amount of numbers together and return their sum:

```
def sum_all_numbers(list_of_numbers):  
    number_sum = 0  
    for number in list_of_numbers:  
        number_sum += number  
    return number_sum  
  
sum_all_numbers([1, 2, 3,])
```

```
## 6
```

A function can also return multiple outputs, e.g. we can return the number of elements in addition to their sum:

```
def sum_and_len_all_numbers(list_of_numbers):  
    number_sum = 0  
    number_len = 0  
    for number in list_of_numbers:  
        number_sum += number  
        number_len += 1  
    return number_sum, number_len
```

To capture the output, we can either assign to a single name, a tuple, or assign both at the same time to different variables.

```
sum_and_len_of_numbers = sum_and_len_all_numbers([1, 2, 3])  
sum_and_len_of_numbers
```

```
## (6, 3)
```

```
sum_of_numbers, len_of_numbers = sum_and_len_all_numbers([1, 2, 3])  
sum_of_numbers
```

```
## 6
```

```
len_of_numbers
```

```
## 3
```

Note that when we defined the function `sum_two_numbers()`, we referred to `num1` and `num2` as *parameters*, while we refer to the numbers we pass to the function call (2 and 5 above) as *arguments*. Although this might sound confusing at first, it is a standard followed by many programming languages so it is useful to get accustomed to this terminology.

5.2.1.1 Function composition

Combining functions is referred to as function composition. This practice allows us to write functions that perform one specific task and then combine them for more complicated tasks, which makes code more readable and easier to debug. By composing a function from the built-in `len()` and `sum()` functions, we can create a more succinct and easier to read version of our previous function `sum_and_len_all_numbers()`.

```
def sum_and_len(list_of_numbers):  
    return sum(list_of_numbers), len(list_of_numbers)
```

5.2.1.2 Positional and keyword arguments

Up until now, our function calls have included just enough arguments to assign one to each of the function parameters. The assignment has been based on the position of the arguments in the function call and therefore these are called positional arguments. We could be more explicit and include the parameter name in the assignment.

```
sum_two_numbers(num1=2, num2=5)
```

```
## 7
```

The arguments are now referred to as keyword arguments and has the advantage that they can be specified in any order.

```
sum_two_numbers(num2=5, num1=2)
```

```
## 7
```

5.2.1.3 Defining default values

Above, we have specified a value for every argument in each function call. This is manageable when functions have few parameters, but it can get tedious for functions with many parameters. Defining default values for select parameters can facilitate working with complex functions by reducing the number of arguments that need to be defined when calling the function and guide users to good default choices for the parameters without requiring in-depth knowledge of each parameter. As an example, we can modify `sum_two_numbers()` to optionally return the two input arguments.

```
def sum_two_numbers(num1, num2, return_input=False):  
    if return_input:  
        return num1, num2, num1 + num2  
    else:  
        return num1 + num2
```

By default the function will work just as previously.

```
sum_two_numbers(2, 5)
```

```
## 7
```

But we now also have the option to return the input numbers.

```
sum_two_numbers(2, 5, return_input=True)
```

```
## (2, 5, 7)
```

Since the arguments are given in the same order as in the function definition, we could have left out the `return_input=` part and just written `True` in the third position.

5.2.1.4 Function documentation

Functions might appear self-explanatory when they are being written, but it is essential that there is proper documentation describing what the function does and what types of arguments should be in the input.

This helps other people who are reading your code and also your future self that will be reusing these functions.

In Python, a function is documented in its docstring, which is a multi-line Python string immediately following the function definition. It is surrounded by triple quotes (either single or double) and can look like this.

```
def sum_two_numbers(num1, num2):  
    '''  
        Add two numbers  
  
        Parameters  
        -----  
        num1: int, float  
            The first number to be added.  
        num2: int, float  
            The second number to be added.  
  
        Returns  
        -----  
        int, float  
            Sum of the two numbers.  
    '''  
  
    return num1 + num2
```

The above docstring convention is referred to as the **numpy** docstring format. There are other conventions, but here we recommended using the **numpy** docstring format since it is easy to read for complex functions with many arguments and used by many integral data science Python packages. It is described in great detail in the **numpy** and **pandas** documentation.

Docstrings constitute the text that show up in the function help message, so it is important that these are well-written and helpful for the reader.

```
help(sum_two_numbers)
```

```
## Help on function sum_two_numbers in module __main__:  
##  
## sum_two_numbers(num1, num2)
```



```
##      Add two numbers
##
##      Parameters
##      -----
##      num1: int, float
##          The first number to be added.
##      num2: int, float
##          The second number to be added.
##
##      Returns
##      -----
##      int, float
##          Sum of the two numbers.
```

This is the same text that is displayed in the Spyder help pane, where it is rendered as markdown for rich display with typefaces, headings, etc.

Spyder includes a convenient function to automatically generate docstring templates. Once you have written the function signature and typed out the first triple quote for the docstring, there will be a small pop-up window that reads “Generate docstring”. Click it or press enter and a `numpy` docstring template will be created based on the parameters in the function signature.

5.3 HOW TO MAKE PROGRAMS INDICATE THAT SOMETHING HAS GONE WRONG?

When performing programmatic data analysis, we need to both ensure that our code runs correctly and that it carries out the intended tasks. The Python interpreter will help us with the first part; if a part of the code is not valid, an error will be raised with a message that helps us trace back what part of the code is not correct.

5.3.1 DUCKTYPING - RELYING ON PYTHON TO DETECT UNEXPECTED BEHAVIOR

In our function `sum_two_numbers()`, we could explicitly check that the input arguments are numerical. However, explicitly checking the type of each input parameter quickly becomes tedious and can make functions less readable, especially as they grow more complex. An alternative

FIGURE 5.1 Rich rendering of the docstring in Spyder

approach is to try to perform the intended operations on the input parameters and rely on that Python will raise an error if they are of the wrong type.

```
sum_two_numbers(5, 'six')
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported opera
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
##   File "<string>", line 18, in sum_two_numbers
```

The raised error contains a helpful message that alerts us to what went wrong; `string` objects cannot be added to `int` objects in Python. If the input variables behave correctly then they probably are of the correct types and no explicit checking is needed. This approach is often referred to as “ducktyping” because it makes assumptions on the type of variable based on its behavior, just like the saying

If it looks like a duck, swims like a duck, and quacks like a duck,
then it probably is a duck.

5.3.2 ASSERTIONS - EXPLICITLY CHECKING FOR UNEXPECTED BEHAVIOR

Ducktyping is useful to catch anything that raises an error in Python, but sometimes we might want to stop the code execution for reasons other than a technical Python error. Examples of this include when you have performed a specific operation that you know should give output of a certain shape and to check if a variable is within an allowed range. The `assert` statement allows us to check if a condition is `True` and stop code execution if it is not.

```
assert 1 == 0
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): AssertionError:
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
print('This is only printed if the assertion above is "True"')
```

```
## This is only printed if the assertion above is "True"
```

Since the condition above is `False`, an `AssertionError` is raised and code execution halts. If the assertion is `True`, there is no output and the next line of code is executed (if there is one).

```
assert 0 == 0
print('This is only printed if the assertion above is "True"')
```

```
## This is only printed if the assertion above is "True"
```

As mentioned above, this is useful if we know that a variable should look a certain way, since we can assert if this is the case and guard ourselves from errors that originate early in the pipeline but could give rise to more cryptic errors that are difficult to troubleshoot later in the pipeline.

It is often helpful to add a clarifying message to the assertion statement, especially as assertions become more complex.

```
x = 1
assert x == 0, 'x is not 0'
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): AssertionError: x is not
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Note that similar functionality can be achieved by explicitly raising an error within a `try` and `except` block, or within a conditional statement using `if`, `elif`, and `else`, but `assert` is a simple and readable way of allowing for manual error checking.

5.4 PACKAGES

5.4.1 INSTALLING PYTHON

There are several ways of installing Python on your system. One of the most robust and cross-platform compatible is to install the Anaconda

Python distribution, which is available for Linux, macOS and Windows. Choose to download the Python 3 installer unless you need to work with Python 2 for a specific reason.

5.4.2 USING PYTHON

When the Anaconda installation has finished, Python is accessible by running `python` in a terminal (on Windows, use the `Anaconda Prompt`). Anaconda also includes graphical interfaces for interacting with Python, which can be invoked by running `spyder` or `jupyter-lab` from the command line, these are covered more in detail elsewhere in this book.

5.4.3 WHAT IS A PACKAGE?

A package is essentially a few Python scripts in a specific directory structure coupled with installation instructions for the computer. Python packages can come from various sources. Many that you will use are part of the standard distribution, but packages can be created by anyone and there are thousands of that can be downloaded and installed from online repositories. Note that you will sometimes hear packages referred to as “modules”. The two words are often used interchangeably. Technically, a package is a folder that contains modules (scripts), which in turn contains functions (code).

Certain functionality that is considered essential for the Python programming language is available wherever Python is installed. This is the Python Standard Library, which is [maintained by the core language team][pep13]. Anything you see within the [Python documentation pages][py-docs] is part of the standard library. Other highly useful, but often more domain specific functionality can be accessed separately in the form of Python packages from third party developers.

People around the world have created packages for Python and made them freely available for others to use resulting in one of the richest package ecosystems for any programming language, with packages for web design, prose writing, game development, and data science (to name just a few). Since there are so many packages available, it is not feasible to include all of them with the default Python installation (it would be as if your new phone came with every single app from the app/playstore preinstalled). Instead, Python packages can be downloaded from central repositories online and installed as needed. The two main repositories

are the Python Package Index (PyPi) and the Anaconda package repository. Instead of navigating to these sites with a web browser, downloading the desired packages and installing them manually, there are so called package managers that automate these processes. The package manager for PyPi is called **pip** and the package manager for Anaconda is called **conda**. To install a package either of these can be used, below is an example of what to type on the command line to install the numerical python package **numpy**.

```
conda install numpy
```

```
pip install numpy
```

Uninstalling a package is equally simple

```
conda remove numpy
```

```
pip uninstall numpy
```

Since we have downloaded the Anaconda Python distribution, we will predominantly be using the **conda** package manager. Installing packages with both interchangeably works, but it is recommended to stick to one as much as possible. The Anaconda team has already bundled many commonly used packages together with their Python installer, including most of the common data science packages, such as **numpy** and **pandas**.

Now that you know how to access many of the world's best data science packages right from your terminal, let's see how we can use them!

Pro tip: Some packages are not available in the default Anaconda repositories. User contributed packages are available in Anaconda "channels", use **anaconda search -t conda <package name>**, to find a channel with the desired package. To install this package, use **conda install -c <channel name> <package name>**. The conda forge channel has many of the packages not in the default repositories.

5.4.4 IMPORTING PACKAGES

Many of the functions that we use regularly, such as **print()** and **len()** are part of Python's Built-in functions. These are the functions that are

always available, but often our programs will need tools beyond these built-ins. Importing in this sense means that you are asking your program to “opt-in” to using that package’s content.

The syntax for importing content is the same for standard library packages, third party packages, and local custom files. The lack of difference in syntax means that code can become confusing if you aren’t extremely conversant in many of the packages.

This is where good documentation comes in.

Import statements are generally found at the beginning of script or notebook files.

Note that many of these examples will talk about importing functions, but there are several other types of content that can be imported from a package.

There are two main ways of importing content:

1. `import <package_name>`: Import the entire package, giving you access to the package content by way of the package’s name. E.g. `<package_name>.<function_name>()`.
2. `from <package_name> import <function_name>`: Import the entire package, but only define variables for the functions you list explicitly. This gives you access to the imported content without having to specify the package name. E.g. `function_name()`.

Each of these methods has a way of renaming the package or content name to shorten it or in case there are multiple with the same name and you need to avoid namespace clashes. This is done using the `as` keyword, which is described in detail later in this chapter.

Any installed package can be accessed by typing `import <package_name>` in Python, e.g.

```
import numpy
```

After importing a Python package, we can access any of its functions, by first writing the followed by a period and then the function name, e.g.

```
numpy.mean([1, 2, 3])
```

```
## 2.0
```

You can think of this as navigating to the numpy menu in a GUI software, and then clicking the function you want. You don't need to recall every function name by hard, pressing the TAB key after the period, will bring up all available function and intelligently filter them as you type out more letters, try it! When you start getting familiar with typing function names, you will notice that this is often faster than looking for functions in menus.

It is common to give packages shorter nicknames, which are faster to type. This is not necessary, but can save work in long files and make code less verbose so that it is easier to read:

```
import numpy as np  
  
np.mean([1, 2, 3])
```

```
## 2.0
```

Using this syntax makes the variable `numpy` not defined in the current namespace.

We could also import the mean function directly. Be careful, importing functions this way may cause name clashing and make your code harder to read because it isn't clear where that function came from.

```
from numpy import mean  
  
mean([1, 2, 3])
```

```
## 2.0
```

And even give it a nickname.

```
from numpy import mean as mn  
  
mn([1, 2, 3])
```

```
## 2.0
```


Which of these you use is up to you, but it is common to follow the conventions established by the library's authors, which for `numpy` is `import numpy as np` and use functions via `np.<function_name>`. Following community conventions means that your code will align with common documentation resources and will be understandable to other users of that tool.

One thing to avoid is to import everything from a package, e.g. `from numpy import *`. If this is done with every package, it is almost guaranteed that the same function name will be available from more than one package and it will be difficult to keep track of it if you are using the `mean` function from `numpy` or from another package that you also imported everything from.

Subpackages and their functions can be imported via the dot syntax.

```
from numpy.fft import fftfreq
```

Packages and subpackages might sound complicated to keep apart. It can be helpful to understand that they are folders and files in specific directory structure. Considering only the `numpy` packages we have mentioned so far, the directory structure would look like this.

```
numpy-folder/  
  script-with-the-mean-definition.py  
  fft-folder/  
    script-with-the-fftfreq-definition.py
```

5.4.5 INSTALLING PACKAGES

After Anaconda has been installed on your system, you can use the command line `conda` package manager or the GUI-driven `anaconda-navigator` to install Python packages. For comprehensive instructions on both of these, refer to the official documentation. Brief step-by-step instructions to get up and running with `conda` follow.

1. To install a new Python package from the Anaconda repositories, simply run `conda install <package name>` in a terminal. You can also use the `pip` package manager, but it will be easier to keep track of packages by sticking to one installation method.

2. Some packages are not available in the default Anaconda repositories. User contributed packages are available in Anaconda “channels”, use `anaconda search -t conda <package name>`, to find a channel with the desired package. To install this package, use `conda install -c <channel name> <package name>`. The conda forge channel has many of the packages not in the default repositories.

5.5 HOW TO GET HELP ONLINE

When reading the built-in Python help is not sufficient, there are several online resources that can be helpful. One of the most commonly used resources for data science is the Stack Exchange network which offers Q&A sites both for programming related topics via Stack Overflow, as well as statistics and machine learning via Cross Validated.

If nothing relevant can be found after searching the many existing questions and answers on these sites, it is appropriate to ask a new question! Stack Overflow has detailed instructions on how to create a minimal reproducible example when asking a question to increase the chances that the question receives a specific and helpful answer. The key principles listed on the website recommend that an answer follows these guidelines:

- Minimal – Use as little code as possible that still produces the same problem
- Complete – Provide all parts someone else needs to reproduce your problem in the question itself
- Reproducible – Test the code you’re about to provide to make sure it reproduces the problem

An additional benefit of reducing the problem into this format is that you might discover the error in the process! This process helps narrowing down exactly which region of the code is failing and the act of explaining the problem often reveals a solution before any replies have come in (commonly referred to as rubber duck debugging).

5.6 KEY POINTS

- Functions allow partitioning of code into segments that perform specific tasks. They make code easier to write, read, troubleshoot, and reuse.

- Functions can have default values for certain input parameters, which allows these parameters not to be explicitly set every time a function is called.
- Ducktyping relies on the Python interpreter to raise meaningful errors when an illegal operation tries to execute.
- Assert statements can be used to explicitly check the truth value of a condition and halt code execution if needed.
- When asking for help online, it is critical to include a minimal reproducible example so that the question can receive suitable replies.
- Packages makes it easy to reuse and share code in an organized and automated manner. These can be installed with package managers.

5.7 EXERCISES

1. Write a function that returns the mean of exactly two numbers. Start with the code we used to create the function `sum_two_numbers()` above.
2. Write a function that returns the mean of any amount of numbers. Start with the code we used to create the function `sum_and_len_all_numbers()` above.
3. Write a function that takes a string as its input and returns a tuple with the first and last character of the string.
4.
 - a. Write a function that takes a string as its input and can return any single character. Which character is returned should be determined by an integer passed to a parameter called `character_index` which has the default value of 0.
 - b. Expand on the previous function by including an assert statement that checks if the integer given to the `character_index` parameter is within the length of the string.
 - c. Describe the pros and cons of the ducktyping approach in 4a and the explicit assert approach in 4b.

6 Publishing

6.1 QUESTIONS

- How can I share my work on the web?

6.2 WHY SHOULD I SHARE MY WORK ON THE INTERNET?

A key part of any project is communicating what you have learned. You've already seen how to create documents to communicate your work with Jupyter notebooks, and how to host whole projects on GitHub. This chapter is about sharing your work through a webpage on the internet. Some advantages of sharing your work on a webpage include:

- It's easy for your visitors—they just need to click on a link and see your work. They don't need to know anything about Jupyter Notebooks, HTML, or GitHub.
- It provides a visually friendly and customizable landing point for people interested in your project. You can easily point them to the GitHub repo if they want more details.
- It's easy for you to make updates and those updates are immediately available to any visitors—you don't have to re-send anyone any files.

By the end of this chapter, you'll be able to take an Jupyter Notebook that lives on your computer and share it with the world through a link to your own webpage.

6.3 WHAT DOES IT TAKE TO GET A WEBPAGE ONLINE?

To understand what it takes to get a webpage online, it helps to understand roughly what happens when you point your browser at a web address. When you point your browser at an address, for example <https://merely-useful.github.io/py-publishing.html>, the following things happen:

1. The URL `https://merely-useful.github.io` points to a location on another computer where the relevant files for this website reside. This computer is known as the website host or server, since it *hosts* the webpage files on its local storage drives and *serves* them to us via the internet.
2. The browser requests the file `py-publishing.html` from the host.
3. The browser reads the contents of `py-publishing.html` and displays it for you in the browser window.

In reverse order this process also describes what you need to do to get your own website online:

1. You need an HTML file that describes what people should see on your page.
2. You need to host the HTML file on a computer on the internet.
3. You need a way to associate a URL with the address of your host.

In this chapter, you'll learn how to get your work online by creating HTML files using Jupyter notebooks (#1 above) and how to host your work in a GitHub repository (#2). The third step will be handled by a GitHub service called GitHub Pages. By following the conventions that GitHub Pages expects, you'll be able to make a webpage for any of your repositories available at: `http://{{your_username}}.github.io/{{repo_name}}`.

6.4 HOW DO I GET MY WORK ON THE WEB?

6.4.1 A STARTING POINT

FIXME: revisit this later when more content is fleshed out. Maybe there will be a repo we can rely on all learners having that we can start from.

In practice you'll probably start thinking about a website once you've already done a lot of work on a project—your project will already have some analysis documented in Jupyter notebooks, be in version control and hosted on GitHub. However, so that we can work with a specific example, you'll set up a project in this section that is less developed than where your project might be when you start thinking about making a website.

At the end of this section, you should have a project folder with an example report in `report.Rmd`. This project should also be on GitHub at `https://github.com/{{your_username}}/sharing-work`, where `{{your_username}}` should be substituted with your GitHub username, e.g. mine is at `https://github.com/cwickham/sharing-work`.

Let's start by creating a GitHub repository called `sharing-work`, by following illustrated steps in the GitHub guide.

Open JupyterLab and create a new Jupyter Notebook ("File -> New -> Notebook", or the Python symbol on the launcher page) Rename this file to `report.ipynb` and pair it with an `.Rmd` file using JupyterText. The pairing to an `.Rmd` file is not necessary, but enables a more human-friendly version control workflow.

Change the cell type of the first input cell from code to markdown by using the dropdown menu in the toolbar. Create a markdown title by typing `# Sharing work` in this cell. Create another markdown cell and type "This is a test repo for GitHub pages". Commit the changes to the `report.Rmd` file and push your commits to GitHub. As explained in the JupyterText section, you do not need to commit `.ipynb` files unless GitHub to render the notebook output. This is our starting point.

FIXME: Would it be better to get to this point by getting learners to fork a repo, then "New project -> From version control" in RStudio? Except forking isn't in the plan for the Version Control section.

6.4.2 HTML FILES

HTML is the language of webpages. JupyterLab makes it easy to turn your notebook into an HTML page by clicking "File -> Export Notebook As -> Export Notebook to HTML". Now you will have a file name `report.html` the same directory as the notebook files. HTML is a plain text format with markup tags to indicate how web browsers should display the text. If you open this file in a text editor, you will see the contents of the file. If you open it in a web browser, your browser will read, interpret, and display the HTML for you. You may notice the address bar in your browser looks something like:

```
file:///Users/wickhamc/Documents/Projects/sharing-work/report.html
```

Just as `https://` is a signal that a file resides on a remote server computer, `file://` is a signal that a file lives on your computer locally. In

spite of having “web” in their name, web browsers perfectly displays local files as long as they are in a suitable format, such as HTML. However, you couldn’t give this local address to someone else and expect it to work, because they don’t have this file on their computer.

The HTML produced by JupyterLab is completely self-contained, the browser needs no additional files to display the page as you see it now. So if you email the file `report.html`, your recipients could open it their browser and see the same result. However, our goal will be to put this HTML file on the web, so you can share a link to the file instead of the file itself. You’ll start by having this file accessible at the link `https://{{your_username}}.github.io/sharing-work/report.html`, then learn how to have it displayed with the shorter link `https://{{your_username}}.github.io/sharing-work`

6.4.3 SETTING UP YOUR REPO TO HAVE A WEB PAGE

Our goal is to get the report that is currently living in `report.html` displayed when a visitor heads to `https://{{your_username}}.github.io/sharing-work`. For this we can use GitHub Pages, which is available for any GitHub repository. First create a folder named `docs` and move `report.html` into this folder, the content of this folder is what will be displayed online. Your online repo structure should now look like this (remember that the `.ipynb` files are not committed and pushed to GitHub, they only exist in your local folder).

```
docs
  report.html
report.Rmd
```

To activate GitHub Pages, navigate to your GitHub Repository in the browser and head to the “Settings” tab.

Scroll to the “GitHub Pages” section. Activate GitHub pages, with source set to “master branch /docs folder”. You should see a message that your site is now live at: `https://{{your_username}}.github.io/sharing-work`

Try visiting: `https://{{your_username}}.github.io/sharing-work/report.html`. You should see your report. Congratulations you have a webpage!

6.4.4 GETTING A DEFAULT PAGE TO DISPLAY WHEN PEOPLE VISIT THE PROJECT SITE

You could send people the link, `https://{{your_username}}.github.io/sharing-work/report.html`, but it is often nicer to send them the shorter version without the file name: `https://{{your_username}}.github.io/sharing-work`. You can try this now, but it won’t work—you’ll see a message in your browser like: “404- File not found”. This shorter URL points to a directory as opposed to a file. By default, when a server receives a request for a directory, it looks for a file to display with a default name—usually `index.html`. In your case there is no file called `index.html` so there is nothing to display.

If you would like the contents of `report.html` to be displayed as the homepage of your project, rename `report.ipynb` to `index.ipynb`. You’ll then need to regenerate the HTML file, commit it, and push your changes. For work that is communicated easily in one page, this would be a good option.

Alternatively, you might have a different page as the default page—one that summarizes the project and then links to other more detailed pages. You’ll see how to do this over the next few sections. To get started create a new Jupyter Notebook paired with an `.Rmd` file. Create a markdown cell with the following content.

```
This is my project...
```

- [Read my report](#)
- [Visit this project on GitHub](#)

Rename the file to `index.ipynb`, export as HTML to `docs/`, commit `index.html`, and push your changes. Your repo should now look like:

```
docs
  index.html
  report.html
report.Rmd
index.Rmd
```

Now when you visit `https://{{your_username}}.github.io/sharing-work` you should see:

Note that it might take some time for the webpage to refresh automatically and detect the new index file. You can trigger this manually by going to GitHub Pages settings and changes the source branch to something else than `docs/` and then back again.

6.4.5 WHAT DOES IT TAKE TO GET YOUR WORK ON A WEB-PAGE?

To sum up the process above, in its most minimal form, to have a webpage at `https://{{your_username}}.github.io/{{repo_name}}`, your repo at `https://github.com/{{your_username}}/{{repo_name}}` needs to:

1. have an `index.html` file in the `docs` directory
2. have GitHub Pages activated in repository settings with source set to “master branch /docs folder”.

Be aware that everything inside the `docs` folder is now public, even if your repository is private.

You might have noticed this book lives at a GitHub Pages URL without a repo name—there is nothing after the `.io` in `https://merely-useful.github.io`. Both GitHub organizations and individuals can make use of this shorter address (`merely-useful` is an organization rather than a user, so this is an organization site). There are a few differences between what you’ve learnt so far and the process of setting up a user site without a repo name (at `https://{{your_username}}.github.io`). First, you need to name your repository in a specific way—it must be called `{{your_username}}.github.io`. Second, user sites don’t use the `docs/` folder—you put your HTML files at the top level in the repo. And third, you don’t have to change any settings with user sites—GitHub will recognize the repo name and automatically serve it at `https://{{your_username}}.github.io`.

6.4.6 EXERCISE: CUSTOMIZE INDEX.RMD

- Edit `index.Rmd` to have your name as the author.
- Export `index.Rmd` to HTML verify your changes in the browser, then commit and push them.
- Visit `https://{{your_username}}.github.io/sharing-work` to check the updated site.

This is the workflow for making changes to your webpage. Make edits locally, and export the notebook to check them. Then commit and push to make those changes visible on the web.

6.5 HOW DO I LINK TO OTHER PAGES, FILES OR IMAGES?

6.5.1 LINKING TO OTHER PAGES

To create a link to another page in markdown file you use the syntax:

```
[text to display](url)
```

Once exported to HTML, only `text to display` will be visible, and clicking on the text will take a viewer to `url`. For example, to add a link to my GitHub repo I might add the following line to `index.Rmd`:

```
Visit [my github repo](https://www.github.com/cwickham/sharing-work)
```

Which when exported to HTML renders like:

Visit my github repo

This is an example of an **absolute** URL. Just like when you specify file paths on your own computer, URLs can be both absolute and relative. An absolute URL describes a file location starting from and including the domain name. For instance, the absolute URL that points to `report.html` in my repo is `https://cwickham.github.io/sharing-work/report.html`.

Relative URLs are *relative* to the current HTML file. So, for instance if you are viewing `https://cwickham.github.io/sharing-work/index.html`, a relative link to my `report.html` would be `report.html` since this file is at the same level as `index.html` in my website structure. For pages created using this GitHub Pages workflow, your website structure is the same as the file structure in your `docs/` folder.

To add a link to `report.html` in `index.Rmd` I would add a line like:

```
See the [full report](report.html)
```

You should use relative URLs to reference any of **your** files (i.e. those in `docs/`). That way if you ever rename your repository, move it, or use a different hosting platform, your links will all work without changes. You must use absolute links for files that reside elsewhere on the internet.

6.5.2 EXERCISE: RELATIVE LINKS

Imagine your `docs/` folder had the following structure

```
docs
  index.Rmd
  index.html
  diagrams
    workflow.png
  reports
    jan.Rmd
    jan.html
    feb.Rmd
    feb.html
```

Using a relative URL, how would you refer to:

- `jan.html` from `index.html`?
- `feb.html` from `jan.html`?
- `workflow.png` from `index.html`?

6.5.3 EXERCISE: ADD LINKS TO INDEX.HTML

Add to `index.Rmd`:

- a link to `report.html` using a relative link, and
- a link your GitHub repository using an absolute link.

6.5.4 LINKING TO SECTIONS WITHIN A PAGE

URLs can also refer to places inside the current page, most usually to another section. In Markdown you've seen how to create headings using `#`. For example, an Appendix subsection might be:

```
## Appendix
```

If we want to link to this section from elsewhere, you prefix the section name with a single `#` in the URL. For example, if this section is in `report.Rmd` and you want to link to it elsewhere in `report.Rmd`, you could use:

See more details in the [\[appendix\]\(#appendix\)](#)

The URL `#appendix` is interpreted as the heading with ID `appendix` in the current page. Jupyter Notebooks creates IDs for all sections (and subsections) automatically, by converting to lower case and replacing spaces with dashes (-). But, you can explicitly set IDs too, by adding the ID with the `#` inside curly braces after the section heading. For instance you might prefer the shorter `appen` ID. You need to set it where the heading occurs:

```
## Appendix {#appen}
```

Then you can link to it using this shorter ID elsewhere:

See more details in the [\[appendix\]\(#appen\)](#)

You can also use this strategy to link to sections in other pages by including the relative URL first. For instance, to refer to this “Appendix” section from `index.html` you could include in `index.Rmd`:

Some gory details of the analysis can also be found in the [\[Appendix of the report\]\(report.html#appen\)](#)

6.5.5 EXERCISE: ADD AND LINK TO A SECTION IN REPORT.RMD

Add a new section to `report.Rmd` and include a link to it in `index.Rmd`. You’ll need to export both `report.Rmd` and `index.Rmd`, and commit and push the HTML files to check your work.

6.5.6 INCLUDING IMAGES

Most of your images will likely be plots generated by code in the notebook and thus automatically included in the exported HTML. If you want to display other images, you use the same syntax you saw in the Markdown section of the Reproducibility chapter. That is, in your notebook you’ll include the image with something like:

```
![[Image caption here.](path/to/image/file.png)]
```

However, the `path/to/image/file.png` should be a relative URL pointing at an image in your `docs/` directory. For example, if you had an image, `me.png`, inside an `images` directory inside your `docs` folder:

```
docs
  index.Rmd
  index.html
  report.Rmd
  report.html
  images
    me.png
```

You could include it in `index.html` by adding to `index.Rmd` the line:

```
![[A picture of me](images/me.png)]
```

Notice the syntax is very similar to adding a link to `me.png`:

```
[[A picture of me](images/me.png)]
```

Including the image displays the image inside at the appropriate place in the current page, *linking* to the image requires a viewer to click the link to see the image.

Your image will be included at full size, but you might find it too large. You can additionally specify some attributes for the image in curly braces immediately following the link. For instance, use the `width` attribute to set the image width, either in pixels:

```
[[A picture of me](images/me.png){width=50} # default unit is px]
```

Or as a percentage:

```
[[A picture of me](images/me.png){width=50%}]
```

6.5.7 EXERCISE: ADD AN IMAGE TO INDEX.HTML

Include an image in your `index.Rmd`. *(If you need an image to include you could always build your own version of an Octocat, GitHub's mascot).*

Don't forget, you'll need to commit both your exported `index.html` and your image.

6.6 EXERCISE: ADD A WEBSITE TO AN EXISTING PROJECT

Add a website to one of your existing project repositories. You'll need to complete the following steps:

- Create a `docs/` directory in your project.
- Add an `index.Rmd` document to the `docs/` folder and export it to HTML to produce `index.html`.
- Commit these changes to git, and push to GitHub.
- Activate GitHub Pages in the repository settings with source set to "master branch /docs folder".
- Visit the site to check it is working.

7 Version Control

7.1 QUESTIONS

- How do I make changes to a file while still saving an original version?
- How can I recover old versions of a file?
- How can I work on the same project on multiple computers?
- What are the pros and cons of putting my project online?

7.2 WHAT IS VERSION CONTROL?

FIXME: Figure - PhD Comics, A Story Told in File Names <http://phdcomics.com/comics.php?f=1323>

When working with files on a computer - documents, scripts, anything involving text - it's common to run into the situation above. There may be information you don't want in the final version, but you want to be able to refer back to previous versions of the file. You may want to work on a document and then start over if it's not going well. It is useful to have a record of changes you've made and the previous versions of your work just for your own psychological reassurance.

Specifically applied to our world of data science projects, here are some examples of when we might want to maintain multiple versions of one file:

- adding a new dataset to our analysis
- constantly modifying or updating the same script
- incorporating comments or feedback from a reader
- keeping backup copies of important files

The multi-file system above is not ideal though! It requires making multiple copies of the same file, even if the changes are isolated to one part of the file. It also isn't always easy to know which file has the information that you want to retrieve.

Luckily, there's a better solution - using a version control system. A version control system is a specific tool that can be used to track changes

in a file or set of files. The version control tool keeps a record of changes made to a file, while still only having one copy of the file on your hard drive. This record of changes consists of snapshots called “revisions” or “commits”. Most of the day-to-day use of version control systems consists of periodically creating these revisions/commits to record the current state of a file or folder, and how it’s changed since the previous revision. Whenever you create a saved revision through a version control system, you have a chance to add a message that provides some information about the file(s) to that point, making it easier to find previous versions of a file.

This chapter will focus on a specific version control tool called “git”. Git is a program that can be installed on your computer and we’ll use it to perform all of the functions described above—creating saved revisions with messages, referencing previous versions of files, and duplicating this revision history across multiple computers.

There are other version control programs that can be used for the same purposes; CVS, SVN, and Mercurial are three examples. We’ve used git in this chapter because it’s commonly used and is compatible with at least two free, online spaces for hosting version controlled work.

We’ll be using a program called GitKraken to make it easier to use git, in the same way that we used the Spyder IDE to help us write Python code.

7.3 CREATING A REPOSITORY

To get started with tracking work, it’s good to think about what scope of information you want to track. Does it make sense to track all of the files of your computer at once? Probably not. It would be better for the version control system to track changes on a per-project basis. In terms of how we have files organized on our computer, that means we want to track changes inside a project folder (and all its sub-folders).

The mechanism used by git to track work is called a “repository” (or “repo” for short) and so for each project where we want to track changes, we need to create a version control repository (in our case, a git repository) associated with the parent folder of the project.

In the GitKraken interface, there’s a button to do exactly that, over on the left: “Start a local repo”. It’s possible to use this button to associate a repository with a pre-existing folder, but it’s simpler to generate a repository with a new folder in one go, so that’s what we’ll do.

1. Click on the “Start a local repo” button.
2. This should open up a three part window, where the left part has “Init” selected, the middle part has “Local Only” and the right has some options to fill in.
3. Fill in a name for your repository in the “Name:” box. It can be anything you want.
4. If you want, choose where to create this repository/folder by selecting a location on your computer in “Initialize in:”
5. Finally, click on “Create Repository”

FIGURE 7.1 Getting Started

Once this process succeeds, a few things should happen. You should see a folder called `test` on your computer’s desktop and the GitKraken window will suddenly have a lot more panes. Just like with Spyder, we’re going to focus on just a part of the window for now - the right hand pane, and the middle section.

At this point, we’ve created a git repository associated with our project folder. Now if we make any changes in that folder (or a sub-folder!), git will pay attention and be able to track it for us. Let’s see what happens if we do this.

7.4 TRACKING CHANGES

As we add and modify files in our project folder, we want to record our changes. This is done by creating “commits” which are like snapshots of the repository at moments in time. By building up a history of many

FIGURE 7.2 Important Panes

commits, each created for a different phase of the files in the repository, you can see what changes have happened over time.

The process of tracking changes in this way has three stages:

1. Create new files or make changes to existing files in the project repository.
2. Choose the files that you want to create a saved version of (called “staging” or “adding”)
3. “Commit” the changes from the previous step, with a descriptive commit message.

To see this in action, create a file called `README.md` in your test repository. You can technically do this through GitKraken, but it would make more sense to create this file in a text editor like Spyder - it should just be a plain text document, saved in the main folder.

```
# Carbon Levels Project
```

```
## Meeting time
```

```
- December 14, 2019  
- 18:00 UTC
```

```
## Collaborators
```

FIGURE 7.3 commit-cycle

```
- Gina  
- Mark  
- Jesus
```

```
## My notes
```

After creating this file, switch back to GitKraken and look at the right side of the GitKraken interface. You should see a few changes in the right hand pane. If you click on “View Change”, the right pane will change to show you the list of files that git knows has changed. In this case, it should have the new `README.md` cycle.

It’s now time to do the second part of the cycle—*adding* or *staging* files to commit. In the GitKraken interface, this can be done by hitting the “stage all changes” button (which will add *all* the changed files to the list to be committed) or you can click on individual files and then click on the “Stage File” button.

Once you’ve done this, the files that are to be added to the commit will be now in the second box of the right pane, the staging area. If you made a mistake and added files you aren’t yet ready to snapshot, you can “unstage” them here. Once you have the list of files you want to stage, write a commit message in the third box. Then click on the commit changes to one file.

We’re sent back to our history mode, with the right pane showing us the latest commit. We’ve gone through the change-and-commit cycle one time. Git has saved information about the state of our `README.md` file in the repository we created in the previous section. Now, if we make additional changes to the `README.md` file, we’ll be able to create new commits, building up a history of changes.

Try making changes to `README.md` under the “My Notes” section and committing them. Create a separate Python script and commit that as well, to see what it looks like to have multiple files in the repository. The best way to get used to the process of using version control to track changes is to run through it a few times.

Some questions to think about as you practice making commits:

- How often should you create new commits after making changes?
- It’s best to create commits fairly frequently, so that you have a good snapshot of your progress. If changes need to

take effect at the same time on a group of files, you should commit them together; if changes are unrelated, it's a good idea to separate them into their own commits.

- What makes a good version control commit message?
 - A good commit message is succinct but meaningful. While many developers can easily slip into commit messages like “@)##(8#@)(!”, it's worth the extra few seconds to type something meaningful, especially for our next task: examining previous versions of files.

7.5 EXAMINING PREVIOUS VERSIONS

We want to keep track of previous versions of a file because sometimes, it's important to go back and either access the previous version in total, or just see the differences between where we started and where we are now. We've used Git to create a list of commits that record versions of a file; now we want to use that history to look back at previous versions of files and compare them with our current version.

Another way to think about this is that instead of just going around the change-and-commit cycle in one direction, we want to be able to reverse direction and look at the past.

In order to compare versions of files, we first need to find the versions we want to compare, from somewhere in our repository's log or history. This is why our commit messages were important! Once you find the commit (or commits) that have the versions you want to reference, it's relatively simple to compare them or revert a file back to its previous version.

If you haven't already, make a few more changes to the “My notes” section of the `README.md` file and commit them, and do this enough times to have 3-4 commits in your central history window.

The history of commits is in the center pane of the GitKraken window. Choose one of the commits you want to examine, and click on it. You'll see the right hand pane show all the information associated with the commit, including the file(s) modified in that commit. (If you want to see *all* the files as they were during a particular commit (not just the modified files), select the desired commit and then click the “View all files” checkbox in the middle-right of the right-hand pane.) Click on a file to see the version saved in the commit.

The file will open in the center pane (replacing the history) and at the top will be a toggle between “File View” and “Diff View”. “File View”

FIGURE 7.4 revert-image

shows the file as it was when the commit was made. If you click on the “Diff View” side, the file will show how it changed since the previous version of the file. Additions are indicated with green highlighting and a plus; deletions with red highlighting and a minus.

You can compare in more selective ways by selecting a whole series of commits, from the commit history, clicking on a changed file, and looking at how the file has changed from the most recent commit you selected to the one preceeding the oldest commit you selected.

Finally, sometimes you may want to return back to a previous version altogether. To do this, select the commit you want to return to from the commit history. Right click on it and then select “Revert commit”. GitKraken will ask you if you want to commit changes immediately. Clicking “yes” will change the file back to a previous version and simultaneously commit it; clicking “no” will simply change the file back, but if you want to keep it, you have to go through the add and commit process yourself.

7.6 ONLINE REPOSITORIES

To this point, we’ve used git features to track changes and compare versions of files in our project repository. An additional way to use git to manage our project is to create a duplicate of our project repository online. As we commit changes to the project on our computer, we can periodically update the online copy so that we have a backup. Sending changes from our computer to an online copy is called “pushing” and if we want to get changes from the online copy to our computer, the command is called “pulling.” In what follows, we’ll set up a blank online repository and push and pull changes to and from it.

We’ll be using Github as the home for the online copy of the project repository. To start, create an empty repository on Github by going to the main Github page and clicking on the green “New” button on the left. You must be logged in for this to work!

You’ll be taken to a screen that gives you some options about your new repository - because we want it to be blank, leave all the options as they are. The only piece you need to fill in is a name for the repository. This doesn’t have to be the same as the folder on your computer, but for clarity, it should be clear that they’re related.

After you click on the “Create repository” button, you’ll be taken to a page with your new blank repository. The link in your browser should be of the format

FIGURE 7.5 remote-image

```
https://github.com/username/repositoryname
```

Copy this link - we'll need it in a moment.

The next step is to link the project repository on our computer with the (blank) online one. To do this, venture into the left pane of the GitKraken Window, where it says "REMOTE". Click on the plus that appears there and a window will open for adding a remote (online) repository. You can give it a nickname in the "Name" box and then paste in the link to your repository for both the Pull URL and Push URL. Click "Add Remote."

The nickname for your online repository should now appear on the left. We want to update it with our local copy—aka, a "push." Click on the push button on the top of the GitKraken screen and use the default options. If you refresh the Github page with your repository, it should now be updated with the information you've committed on your computer.

If you're always making changes on your own computer, you can keep using the commit cycle from before, and periodically push those changes to the remote (online) copy of the repository.

However, it's also possible (and common) that changes are made to the online copy that you'll want to update on your own computer. To test this out, first make a change to the `README.md` file in your Github repository. You can do this by clicking on the file, clicking on the pencil icon on the top-right of the file, making a change, and then clicking the "Commit changes" button at the bottom of the page (Figure 7.6).

There are now changes in the online copy that are not in your local copy. To sync them back up, go back to GitKraken on your computer and click the "pull" button. The history of commits in the center pane should update to include the change you just made online.

Note that we started with a repository on a computer and duplicated it online, but it can go the other way as well. Close the currently open repository in GitKraken. You should be back at a screen with several options, including "Clone a repo". If you click on this button, you can paste in the link of a repository to download from the internet and where to put it on your computer. If you try this with the link to your own Github repository (but change the location on your computer), Git will download the same repository in a different place; if you open it, you'll see that it looks exactly the same as the repository you just closed.

The moral of the story is that you can create as many copies of a project repository as you want, but it's a good idea to choose one of to be "official" version that always has the most-up-to-date copy of your changes.

FIGURE 7.6 Committing

Frequently, this is the online copy, as it's the easiest to access. Anytime you start working on a local copy of the repository, it's a good idea to "pull" from the online copy to make sure you're up-to-date, and then "push" at the end, to make sure your changes are available online.

7.7 CONFLICTS

Even the most careful person will end up making changes in two different copies of the same repository and then trying to sync them together. Sometimes git is able to resolve the differences on its own, but sometimes the two changes conflict and can't be automatically reconciled.

We can create a conflict like this by making changes in our `README.md` file. First, make sure that your local repository and Github repository are synced up - that you have pushed all your local changes to Github and pulled all your Github changes to your local computer.

Then, **on your computer**, make these changes to the `README.md` file:

```
## Meeting time

- December 21, 2019
- 18:00 UTC
```

Commit the changes as we've done before but do **not** push them to Github.

Now, go to your repository on Github, click on the `README.md` file and edit it as follows:

```
## Meeting time
```

- November 4, 2018
- 18:00 UTC

Commit the changes.

Now we have conflicting changes in our repository—one in the Github copy and one on our local computer copy.

We want to sync our changes by either pushing from the copy on our computer to the Github copy or by pulling from the Github copy to our computer. It turns out that we can't do the first option (try it and see what happens), so we'll have to use the second option, of pulling from Github to our computer. When we try this, we should get a few red boxes that pop up saying "Pull failed" and/or "Merge failed".

Note that, when we try to push, we *do* have the option of using a "force push" to override whatever changes we made in Github. If you're absolutely sure that there's nothing in the Github copy you want, you can try this, but it's generally a good idea to pull down the changes and look at the conflict, just to make sure.

Because of the conflicting changes we made, the update of our local repository failed. Git doesn't know which changes we really want and has pushed us into a temporary "conflict" state until we tell it specifically which changes we want.

To do this, click on "View conflict" (should be on the top right of the GitKraken window). Click on the file in conflict (`README.md`). This will open up a different view, where the two conflicting versions of the file are shown side by side. You can check the box picking one version, the other, or both, with the combined output shown in the window below. Once you've picked what you're happy with, click the "Save" button on the top right, close the file by clicking the "X" and finish the update by clicking on "Commit and merge" (Figure 7.7).

FIGURE 7.7 The Commit Process

7.8 USING LOCAL VS REMOTE REPOSITORIES

In the previous section, we set up an online copy of our project repository. This wasn't too risky because our repository didn't have any large file or confidential information. However, for some projects, that will be the case, which is why it's important to think carefully before creating an online repository.

Consider the following before you put it in an online repository:

- Size
 - Github has file size limits; you won't be *allowed* to upload files of a certain size. Projects with large data files often store their data in other places and then include directions of how to access it in the main project repository.
- Privacy - If your data includes identifying information or other confidential features, it should be placed in a location with the appropriate security.

7.9 TO GIT-FINITY, AND BEYOND

Git was originally developed to manage a BIG and complex software project (development of the Linux operating system kernel) so as a version control system, it's also BIG and complex, with a lot of features that aren't needed for most everyday workflows, especially if you're mostly

working by yourself. This chapter has covered what is most essential, and hopefully, most useful. If you find yourself working on more complex projects, especially with multiple collaborators, you may want to learn more about some of git's development and collaboration features. See (book 2?) for more information.

7.10 EXERCISES

1. Abbie has started working on a data analysis project. She has a copy of the main data set, and sent her supervisor a preliminary report based on a script she has been writing. The response to her report was a lot of comments, and an updated version of the dataset. Meanwhile, she had a colleague look at her script and he refactored it into a more conventional coding standard. Where are some areas where Abbie could use version control to track her work?
2. Make a change to `README.md` and commit the changes to the repository. How can you check that the changes were committed?
3. **FIXME:** Screenshot of version history with this series of commits: "Initial commit", "Adding script", "Removing written in filename", "Using new package functions", "Adding multi-file capabilities."

How would you look at the difference between the current script and when it had the input filename written into the script?

1. Manuel has a copy of his main repository on his laptop, work computer, and Github. If he has done work on his laptop while traveling, what steps does he need to go through to update the copy on his work computer? What should he have done before starting to work on his laptop to make this go as smoothly as possible?
2. **FIXME:** Screenshot of message you get when there's a pull/merge conflict. What does this message indicate about the state of the repository? What would you do next?
3. Which of these is/are examples of remote storage?
 - GitHub
 - Google Drive
 - an external hard drive

Name some pros and cons of each option.

7.11 KEY POINTS

- A version control system allows you to record the history of changes to a file and recover older versions when needed.
- Each snapshot that captures information about a change is called a commit.
- Commits are stored in a version control repository, usually associated with a project folder and all its nested sub-folders.
- A version control tool like Git has commands that allow you to make commits, review them, and revert back to previous commits (if needed).
- Make commits frequently when actively working on a project.
- Version controlled repositories can be duplicated online or on other computers.
- A public repository can be duplicated and used to work on any computer.
- Large data files and private information should not be stored on a public version control repository.

A License

This is a human-readable summary of (and not a substitute for) the license. Please see <https://creativecommons.org/licenses/by/4.0/legalcode> for the full legal text.

This work is licensed under the Creative Commons Attribution 4.0 International license (CC-BY-4.0).

You are free to:

- **Share**—copy and redistribute the material in any medium or format
- **Remix**—remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution**—You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions**—You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

B Code of Conduct

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

B.1 OUR STANDARDS

Examples of behavior that contributes to creating a positive environment include:

- using welcoming and inclusive language,
- being respectful of differing viewpoints and experiences,
- gracefully accepting constructive criticism,
- focusing on what is best for the community, and
- showing empathy towards other community members.

Examples of unacceptable behavior by participants include:

- the use of sexualized language or imagery and unwelcome sexual attention or advances,
- trolling, insulting/derogatory comments, and personal or political attacks,
- public or private harassment,
- publishing others' private information, such as a physical or electronic address, without explicit permission, and
- other conduct which could reasonably be considered inappropriate in a professional setting

B.2 OUR RESPONSIBILITIES

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

B.3 SCOPE

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

B.4 ENFORCEMENT

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by emailing the project team. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

B.5 ATTRIBUTION

This Code of Conduct is adapted from the Contributor Covenant version 1.4.

C Contributing

Contributions of all kinds are welcome. By offering a contribution, you agree to abide by our Code of Conduct and that your work may be made available under the terms of our license.

1. To report a bug or request a new feature, please check the list of open issues to see if it's already there, and if not, file as complete a description as you can.
2. If you have made a fix or improvement, please create a pull request. We will review these as quickly as we can (typically within 2-3 days). If you are tackling an issue that has already been opened, please name your branch `number-some-description` (e.g., `20-highlighting-active-block`) and put `Closes #N` (e.g., `Closes #20`) on a line by itself at the end of the PR's long description.

C.1 STYLE GUIDE

We follow the tidyverse style guide for R and PEP 8 for Python as closely as possible but specify some conventions further. We go against the style guides only when it is considered that it will improve clarity.

Specific conventions include:

- `variable_name` (snake_case)
- `function_name` and `method_name` (snake_case)
 - Please do *not* include empty parentheses to indicate a function, as this makes it hard to distinguish a function name from a call with no arguments.
- `folder-name/` (hyphens instead of underscores, trailing slash for clarity)
- `file-name` (hyphens instead of underscores)
- `'string'` and `"string"`
 - We will settle on single vs. double quotes before we publish :-)
- Method chaining in pandas:

```
(dataframe
 .method()
 .method(short_arg)
 .method(
   long_arg1,
   long_arg2))
```

For markdown, we use ATX-headers (`#` prefix) rather than Setext headers (`=/-` underlines), links with `[linkname][tag]` rather than `[linkname](url)`, and fenced code blocks rather than indented blocks.

There are more details for what we recommend for learners in `rse-style.Rmd`. Discuss further in issue #116.

Please note that we use Simplified English rather than Traditional English, i.e., American rather than British spelling and grammar.

C.2 SETTING UP

This book is written in Bookdown. If you want to preview builds on your own computer, please:

1. Follow the instructions for installing Bookdown.
2. Run `make everything` to recompile everything.
 - Run `make` on its own to see a list of targets for rebuilding specific volumes as HTML or PDF.

Please note that Bookdown works best with TinyTeX. After installing it, you can run `make tex-packages` to install all the packages this book depends on. You do *not* need to do this if you are only building and previewing the HTML versions of the books.

D Glossary

Abandonware FIXME

Absolute error FIXME

Absolute path FIXME

Accuracy FIXME

Action (in Make): FIXME

Active listening FIXME

Actual output (of a test): FIXME

Actual result FIXME

Aggregate FIXME

Agile development FIXME

Ally FIXME

Analysis and estimation FIXME

Annotated tag (in version control): FIXME

Append mode FIXME

Application Programming Interface (API): FIXME

Assertion FIXME

Authentic task A task which contains important elements of things that learners would do in real (non-classroom situations). To be authentic, a task should require learners to construct their own answers rather than choose between provided answers, and to work with the same tools and data they would use in real life.

Auto-completion FIXME

Automatic variable FIXME

Automatic variable (in Make): FIXME

Backlog FIXME
Bash FIXME
Beeswarm plot FIXME
Binary code FIXME
Bit rot FIXME
Boilerplate FIXME
Branch FIXME
Branch-per-feature workflow FIXME <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>
Breakpoint FIXME
Buffer FIXME
Bug report FIXME
Bug tracker FIXME
Build tool FIXME
Build tool FIXME https://en.wikipedia.org/wiki/List_of_build_automation_software
Byte code FIXME
Call stack FIXME
Camel case FIXME

Catch (an exception): FIXME
Checking-driven development (CDD): FIXME

Checklist FIXME
Code browser FIXME
Code coverage FIXME
Cognitive load FIXME

Comma-separated values (CSV): FIXME

Command argument FIXME
Command flag FIXME
Command history FIXME
Command option FIXME
Command shell FIXME
Command switch FIXME

Command-line interface (CLI): FIXME

Comment FIXME

Commit hash FIXME

Commit message FIXME

Commit FIXME

Commons FIXME

Competent practitioner Someone who can do normal tasks with normal effort under normal circumstances. See also novice and expert.

Compiled language FIXME

Compiler FIXME

Computational competence FIXME

Computational notebook FIXME

Computational stylometry FIXME

Computational thinking FIXME

Conditional expression FIXME

Configuration object FIXME

Confirmation bias FIXME

Context manager FIXME

Continuation prompt FIXME

Continuous integration FIXME

Corpus FIXME

Coverage FIXME

Creative Commons - Attribution License (CC-BY): FIXME

Curb cuts <https://medium.com/@mosaicofminds/the-curb-cut-effect-how-making-public-spaces-accessible-to-people-with-disabilities-helps-everyone-d69f24c58785>

Current working directory FIXME

DRY (Don't Repeat Yourself) The general principle when programming that it's typically better to define something (a function, a constant...) once and refer to it consistently as a "single source of truth" throughout a piece of software than to define copies in multiple places, if only because then you only have to make any changes in one place. This is useful and helpful principle but should not be thought of as an inviolate rule.

Data engineering FIXME

Data package FIXME

Declarative programming FIXME

Default target FIXME

Default target (in Make): FIXME

Delimiter FIXME

Dependency graph FIXME

Design pattern FIXME

Design patterns FIXME

Destructuring FIXME

Dictionary FIXME

Digital Object Identifier (DOI): FIXME

Directory Tree If the nesting relationships between directories in a filesystem are drawn as arrows from the containing directory to the nested ones, a tree structure develops.

Directory A folder in a filesystem.

Disk Disk refers to disk storage, a physical component of a computer that stores information on a disk. The most common kind of disk storage is a hard disk drive, which is a storage drive with a non-removable disk.

Docstring FIXME

Documentation generator FIXME

Downvote FIXME

Embedded documentation FIXME

Eniac FIXME

Environment FIXME

Error (result from a unit test): FIXME

Escape sequence FIXME

Exception FIXME

Exit status FIXME

Expected output (of a test): FIXME

Expected result FIXME

Expert Someone who can diagnose and handle unusual situations, knows when the usual rules do not apply, and tends to recognize solutions rather than reasoning to them. See also competent practitioner and novice.

Exploratory programming FIXME

Exponent FIXME

External error FIXME

Failure (result from a unit test): FIXME

False beginner Someone who has studied a language before but is learning it again. False beginners start at the same point as true beginners (i.e., a pre-test will show the same proficiency) but can move much more quickly.

False negative FIXME

False positive FIXME

Feature boxing FIXME

Feature creep FIXME

Feature request FIXME

Filename extension FIXME

Filename stem FIXME

Filesystem Controls how files are stored and retrieved on disk by an operating system. Also used to refer to the disk that is used to store the files or the type of the filesystem.

Filter FIXME

Fixture FIXME

Flag variable FIXME

Flag FIXME

Folder FIXME

Forge FIXME

Fork FIXME

Format string FIXME

Frequently Asked Questions (FAQ): FIXME

Full identifier (in Git): FIXME

Fully-qualified name FIXME

Function attribute FIXME

Function (in Make): FIXME

GNU Public License (GPL): FIXME

Git branch FIXME

Git clone FIXME

Git conflict FIXME

Git fork FIXME

Git merge FIXME

Git pull FIXME

Git push FIXME
Git stage FIXME
Git FIXME
GitHub Pages FIXME
Globbering FIXME

Graphical user interface (GUI): FIXME

HTTP status code FIXME
Hitchhiker FIXME
Home directory FIXME
Hot spot FIXME
ISO date format FIXME
Impostor syndrome FIXME
In-place operator FIXME
Index FIXME
Install FIXME

Integrated Development Environment (IDE): FIXME

Internal error FIXME
Interpeter FIXME
Interpreted language FIXME
Interruption bingo FIXME
Issue tracking system FIXME
Issue FIXME

Iteration (in software development): FIXME

JSON FIXME

Jenny (a repository): FIXME

Join (of database tables): FIXME

Kebab case FIXME

Label (in issue tracker): FIXME

Learned helplessness FIXME

Library FIXME

Lint FIXME

List comprehension FIXME

Log file FIXME

Logging framework FIXME

Loop body FIXME

Loop (in Unix): FIXME

MIT License FIXME

Macro FIXME

Magic number FIXME

Magnitude FIXME

Makefile FIXME

Mantissa FIXME

Martha's Rules FIXME

Memory A physical device on your computer that temporarily stores information for immediate use.

Mental model A simplified representation of the key elements and relationships of some problem domain that is good enough to support problem solving.

Method A function that is specific to an object type, based on qualities of that type, e.g. a string method like `upper()` which turns characters in a string to uppercase.

Namespace A way of organizing names of related objects, functions, or variables to avoid confusion with (for instance) common names that might well occur in multiple packages.

Nano FIXME

Ngo FIXME

Not Invented Here (NIH): FIXME

Novice Someone who has not yet built a usable mental model of a domain. See also competent practitioner and expert.

ORCID FIXME

Object An object is a programming language's way of describing and storing values, usually labeled with a variable name.

Object-oriented programming FIXME

Open license FIXME

Open science FIXME

Operating system FIXME
Operational test FIXME
Oppression FIXME
Orthogonality FIXME
Overlay configuration FIXME
Overloading FIXME
Package FIXME
Pager FIXME
Pair programming FIXME
Parent directory FIXME
Parking lot FIXME
Path coverage FIXME
Path FIXME
Pattern rule FIXME
Pattern rule FIXME
Peer action FIXME
Phony target FIXME
Phony target FIXME

Pipe (in Unix): FIXME

Post-mortem FIXME
Pothole case FIXME
Precision FIXME

Prerequisite (in Make): FIXME

Privilege FIXME
Procedural programming FIXME
Process FIXME
Product manager FIXME
Project manager FIXME
Prompt FIXME
Provenance FIXME

Pseudorandom number generator (PRNG): FIXME

Public domain license (CC-0): FIXME

Pull request FIXME

Python FIXME

Raise FIXME

Raise (exception): FIXME

Raster image FIXME

Rebase FIXME

Recursion FIXME

Redirection FIXME

Refactor FIXME

Refactoring FIXME

Regular expression FIXME

Relative error FIXME

Relative import In Python, the importing of a module relative to the current path and thus likely from within the current package (e.g., from `. import generate`) rather than an import from a globally-defined package (e.g., from `zipfpy import generate`).

Relative path FIXME

Remote login FIXME

Remote repository FIXME

Repl FIXME

Repository FIXME

Representation State Transfer (REST): FIXME

Reproducible example (reprex): FIXME

Reproducible research FIXME

Research software engineer (RSE): FIXME

Restructured Text (reST) A plain text markup language used by much Python documentation and documentation tooling.

Revision FIXME

Root directory FIXME

Rotating file FIXME

Rule (in Make): FIXME

SSH key FIXME

SSH protocol FIXME

Scalable Vector Graphics (SVG): FIXME

Script FIXME

Seed (for pseudorandom number generator): FIXME

Semantic versioning FIXME <https://semver.org/>

Sense vote FIXME

Set and override (pattern): FIXME

Shebang FIXME

Shell script FIXME

Short circuit test FIXME

Short identifier (in Git): FIXME

Side effects FIXME

Sign FIXME

Silent error FIXME

Silent failure FIXME

Situational action FIXME

Snake case FIXME

Software development process FIXME

Source code FIXME

Stand-up meeting FIXME

Standard error FIXME

Standard error FIXME

Standard input FIXME

Standard input FIXME

Standard output FIXME

Standard output FIXME

Streaming data FIXME

Sturdy development FIXME

Subcommand FIXME

Subdirectory FIXME

Subsampling FIXME

Success (result from a unit test): FIXME

Sustainability FIXME

Sustainable software FIXME

Symbolic debugger FIXME

Syntax highlighting FIXME

Synthetic data FIXME

Tab completion FIXME

Tag (in version control): FIXME

Tag FIXME

Target FIXME

Target (in Make): FIXME

Target (of oppression): FIXME

Technical debt FIXME

Test coverage FIXME

Test framework FIXME

Test isolation FIXME

Test runner FIXME

Test-driven development FIXME

Three stickies FIXME

Ticket FIXME

Ticketing system FIXME

Tidy data As defined in Wickham (2014), tabular data is tidy if (1) each variable is in one column, (2) each different observation of that variable is in a different row, (3) there is one table for each kind of variable, and (4) if there are multiple tables, each includes a key so that related data can be linked.

Time boxing FIXME

Timestamp (on a file): FIXME

Tldr FIXME

Tolerance FIXME

Transitive dependency FIXME

Triage FIXME

Tuning FIXME

Tuple FIXME

Typesetting language FIXME

Unit test FIXME

Unix shell FIXME

Update operator See in-place operator.

Upvote FIXME

Validation FIXME

Variable (in Python): A symbolic name that reserves memory to store a value.

Variable FIXME

Variable (in Make): FIXME

Vector image FIXME

Verification FIXME

Violin plot FIXME

Virtual environment In Python, the `virtualenv` package allows you to create virtual, disposable, Python software environments containing only the packages and versions of packages you want to use for a particular project or task, and to install new packages into the environment without affecting other virtual environments or the system-wide default environment.

What You See Is What You Get (WYSIWYG): FIXME

Wildcard FIXME

Working directory FIXME

Working memory FIXME

Wrap code FIXME

Wrapper FIXME

YAML FIXME

E Novice Python Learning Objectives

This appendix lays out the learning objectives for each set of lessons, and is intended to help instructors who want to use this curriculum.

FIXME: fill in learning objectives for novice Python

F Novice Python Key Points

FIXME: fill in keypoints for novice Python

Haddock, S., and C. Dunn. 2010. Practical computing for biologists. Sinauer Associates.

FIXME

Noble, W.S. 2009. A quick guide to organizing computational biology projects. *PLoS Computational Biology*. 5. doi:10.1371/journal.pcbi.1000424.

How to organize a small to medium-sized bioinformatics project.

Scopatz, A., and K.D. Huff. 2015. Effective computation in physics. O'Reilly Media.

A comprehensive introduction to scientific computing in Python

Taschuk, M., and G. Wilson. 2017. Ten simple rules for making research software more robust. *PLoS Computational Biology*. 13. doi:10.1371/journal.pcbi.1005412.

A short guide to making research software usable by other people.

Wickham, H. 2014. Tidy data. *Journal of Statistical Software*. 59. doi:10.18637/jss.v059.i10.

The defining paper on tidy data.

Wilson, G. et al. 2014. Best practices for scientific computing. *PLoS Biology*. 12. doi:10.1371/journal.pbio.1001745.

Outlines what a mature research software project should look like.

Wilson, G. et al. 2017. Good enough practices in scientific computing. *PLoS Computational Biology*. 13. doi:10.1371/journal.pcbi.1005510.

Outlines what a “good enough” research software project should look like.

Yenni, G.M. et al. 2019. Developing a modern data workflow for regularly updated data. *PLOS Biology*. 17:e3000125. doi:10.1371/journal.pbio.3000125.

FIXME