# Computer skills for graduate students

Kevin R Thornton

2021-01-04

# Contents

 

 

# Chapter 1

# Preface

This book is my attempt to provide a "textbook" for a graduate-level class on "bioinformatics" skills for PhD students that I team teach with Tony Long at UCI. The course is the second in a series. J.J. Emerson teaches the first quarter, which is loosely arranged around Vince Buffalo's book (Buffalo, 2015).

The contents of the book will be terse and idiosyncratic in places. The goal is to provide readings for the first half of the class, which can be broadly defined as "skills for reproducible computational research".

# Chapter 2

# Introduction

This book seeks to introduce "typical" early-stage PhD students in the biological sciences to computer skills that will likely be necessary for them to complete their research. By "typical", I mean someone who was probably a biology major as an undergraduate who now finds themselves doing research in a field where they have to write "scripts" or "programs" to get daily tasks accomplished.

In my corner of research at UC Irvine, I cannot imagine a student being able to complete their dissertation without becoming proficient in languages such as Python and/or R. These programming languages are the subjects of Chapters 6 and 7, respectively, after a brief general introduction to programming concepts in Chapter 5. Further, to make one's life a bit easier, there must be some way to change code that doesn't involve copying files, editing, copying again, and eventually losing track of which version is the current one, or even the one that works. An easier path does exist, in the form of "version control" software, such as git. It is becoming more common to require that code used for research be made publicly-available, meaning that students need to be familiar with the various websites offering free services for code sharing. These topics are covered in Chapter 8.

Please read the chapter on objectives and expectations for the course, Chapter 3. Bear in mind that this is a single ten week course on a huge topic. Thus, we cannot do everything. In fact, we have to completely punt on a lot of interesting and fun stuff! Where possible, I mention such things and note that they are *BTS*, or beyond the scope of this course.

It is useful to keep in mind the ultimate goal of our research, which is to communicate it. This communication will take on many forms, from making plots (or tables) for yourself, for lab meeting, for a meeting with your advisor, for presentation at a conference, for publication, etc.. The "quality" required (in terms of the clarity and technical detail of the presentation) may differ for each of things, but one constant remains throughout. You need a *reproducible* way of going from "raw data" to the final "report". The only reliable way to do this is to generate a pipeline of scripts that can take you from start to finish. Yes, this means no Adobe Illustrator, etc., as there is no way that "future you" will remember which mouse clicks were necessary. In fact, one of the best motivations for generating reproducible computational work flows comes from pure self-interest! As other life events (TA-ing courses, etc.) cause you to walk away from your research, you need to be able to return to your project in the future and quickly get back up to speed. The topics covered in the later chapters will point you to concepts and techniques that you will likely find useful in documenting your research progress as well as in writing your dissertation.

# Chapter 3

# Learning objectives for this class

## 3.1 Expectations for a "hackathon"-style course

The majority of this course is a "learn by doing" or "hackathon" experience. During the "lecture", I'll introduce a topic or set of topics. In the "lab", you will work on problems related to these topics. Given that we cannot possibly teach you everything via the lectures, a lot of the learning comes from having to solve problems in real time in lab. Thus, you'll get out what you put in.

To make progress, you'll need a computer capable of doing the labs (see Appendix A). We also expect that you are proficient in the material covered in the prerequisite course taught by J.J. Emerson. Although we do let students in who have not taken the prerequisite, we reserve the right to monitor your progress. If someone is taking up more than their fair share of the instructor's/TA's time due to a lack of background, then that person may be asked to leave the course.

Briefly, the topics covered in the previous class are:

- **TBD**

## 3.2 What you will not get from this course

When we poll students about their interests in "bioinformatics", the message is that we need a full curriculum spread over two years in order to satisfy everyone. Given that such a curriculum is not possible, we have to distill things down to the "elements" of computational skills for graduate students in the biological sciences. Thus, we will not teach you how to program. We probably will not teach you how to analyze your data. We will not teach you the detailed ins and outs of `git`, `ggplot2`, or some specific piece of bioinformatics software and what all of its options mean. Such topics are *beyond the scope* (BTS) of this course. Further, if we spent time teaching you how to use a specific RNA-seq data aligner, then we may be doing you more harm than good, as that tool may be out of date/fashion by the time you are writing your first paper. The same is true of programming languages. While I find it hard to believe that Python will not widely used in ten years, we were all saying the same thing about `perl` when I was a graduate student. However, `perl` has fallen far from its former position as **the** programming language for bioinformatics, having been largely replaced by Python and/or R about ten years ago. In fact, one of the issues that hampers training graduate students is that your advisors tend not to keep refining their own skill sets throughout their careers, making it difficult to train students starting their own research careers in an era when the languages and idioms that the PI knows are no longer in wide use. (The same is true of lab techniques, although there is more of a bias in the life sciences for staying familiar with the latest in that area.)

## 3.3 What you should expect from this course

By the end of the first half of this course, you should be able to perform all of the following tasks:

- Create `git` repos and commit changes to files. You should be able to `pull`/`clone` to/from multiple sources. For example, you should be comfortable `push`ing from your laptop to `GitHub` and then `pull` from `GitHub` to HPC.
- Submit single and array jobs to HPC.
- Use Rstudio and/or Jupyter (or Jupyter Lab) to generate "reports" of your research.
- Manage dependencies using `conda`. You should know how to create environments and how to "dump" the contents of an environment to a file.
- You should know what tools are out there for preparing figures for various sorts of research communication tasks, and have some sense of the strengths and weaknesses of each.

More generally:

- You should have an expanded awareness of what Python and R can do for you.
- You should know what you don't know, but know how to figure it out and/or get help. This is perhaps the most important outcome!
- You should be better able to search the web for help.
- You will hopefully develop the habit of starting any new task with a `git init` command!

 

# Chapter 4

# Open source software and communities

It is likely that most of the software that you use for your research is *open source software*, or OSS. The primary feature of OSS is that the source code is available. This availability is key for finding bugs, fixing bugs, distributing the software, and allows one to use, study, and modify the product. A corollary feature of OSS is that it is often available free of charge.

Open source software is the product of open source communities, which is something that many people don't take the time to think about.

## 4.1 Volunteer culture

Most OSS is developed, maintained, and supported entirely by volunteers. In relatively rare cases, individuals may be paid by companies to work on OSS, but most of the time, the work is being done by professionals in their "spare" time. It is important to acknowledge the volunteer status of the contributors for many reasons:

- It is amazing when the product is very high quality. Linux, which is the OS for something like 90% of computers worldwide, is almost all-volunteer. R and Python, too, at least in the early days.
- When something isn't working as you expect, or the documentation is lacking, then this too is a side-effect of the volunteer labor force. A lot of OSS software had its origins in solving a particular problem before being released to the public. If the work became popular after its release, the resulting larger user base is guaranteed to find weaknesses. In many cases, including for academic software, it becomes nearly impossible for one person to support such a large user base. Thus, it becomes important that the community contribute back.
- It becomes important to learn how to interact with the community behind various OSS works. I am not saying that toxic behavior found in some communities is acceptable. Rather, there are specific modes of communication used for discussing specific works, which is the topic of the next section.

## 4.2 How to ask for help, report problems, etc.

If you have a question about an OSS work, then you need to know where to ask it, and *how* to ask it. The *how* is especially important when you have a problem getting something working.

There are many ways for you to interact with OSS developers, including:

- GitHub issues

- Gitter
- IRC
- Mailing lists

It is important to note where to go, and to recognize that it varies from product to product. Ideally, the documentation will note the appropriate venue. Note that Stack Overflow is not on the list. I am not aware of any OSS developers that monitors SO for questions about their projects. But I am aware of many instances where questions on SO were cross-posted to the "official" help channels, which is often viewed as not okay. Cross posting is problematic because the answer often only appears in one place, making things confusing for those who come later.

The *how* to ask a question is very important when you cannot get something to work or you think there is a bug in a piece of software. No one can help you if your question is vague or if it requires installing a ton of software and having access to your raw data in order to see the issue. What is needed here is a *minimal reproducible example*, or MRE. The goal here is to construct a self-contained example that fails and send that to the author. Yes, this takes more time than just saying, "I get an error on my data", or "This 300 line R script with your tool in the middle isn't working", but the people volunteering their time to work on this project have zero incentive to help out when issue descriptions are not well done. So, if you want their help, then you probably have to help them out a bit, too.

The act of creating an MRE is often very useful. Sometimes, it does show that there is indeed a bug of some sort in the tool that you are using. Other times, you find bugs in your own code that were the real cause of the problem.

Please note that most OSS developers *welcome* the report of issues with their projects, especially when they are well-crafted. Most people want their work to be useful, and do sincerely want to help, even if they have no time, no financial incentive, etc..

## 4.3 Challenges of developing OSS software

Siepel (2019) gives a thoughtful overview of the problems facing software development in the life sciences. While the focus is on genomics, the problems are more general than that, as many of the incentive structures in academia apply more broadly.

 

# Chapter 5

# Overview of programming concepts

This chapter provides an informal overview of "programming". Basically, I need to define some terms so that we are all on the same page moving forwards. This book has no interest in the debates over different programming paradigms: object-oriented vs functional vs whatever. Rather, we take a practical approach and move on as quickly as possible.

## 5.1 What is programming?

Programming is writing down a bunch of instructions into a file so that a computer can execute them. There really isn't much more to it than that. Our computers exist to execute the logic that we write down. If we do a good job, our lives are easier. If we make a mistake, then we get the wrong answer. (Telling right from wrong answers coming out of programs is a key skill for you to develop during your PhD.) If we make a really bad mistake, then we may cause harmful side effects like accidentally deleting data or crashing the machine. That said, don't be afraid of mistakes–they are an important part of the learning and research process.

### 5.1.1 How to learn programming.

Simply put, you will learn the most by doing it. However, another important way to learn to code is to read code. Specifically, you want to read code that emphasizes the idiomatic way to do things in your language of choice. Learning the "R" way to do a task or the "Pythonic" way will help you move from writing code that may be ugly but gets the job done to readable and efficient code that you can reuse later. It is helpful to think of coding as writing, and to remember that writing is re-writing.

If you are entirely new to coding, then I recommend checking out the lessons available from Software Carpentry.

### 5.1.2 Programming at different levels: scripts vs programs

In order to be practical, let's break programming up into two types of tasks. The first is writing short, focused pieces of code to accomplish a specific task. For example, a `bash` script to automate a set of steps to go from raw data to a final set of results. Another example would be writing the code to make a figure or table from the output of the previous `bash` script. These tasks are often implemented as one-off pieces of code that are used for a project and then forgotten about. (How not to forget about these works, by archiving them appropriately, is the topic of Chapter 8). Colloquially, these short programs are referred to as "scripts" and writing them as "scripting". This nomenclature is imperfect, however.

The second type of task is to write a general tool for the research community that accomplishes a specific task. For example, a program to align short-read sequence data to a reference genome (Li and Durbin (2009),Li (2013)) or a program to simulate genetic data under explicit evolutionary models (Kelleher et al., 2016). A related task involves writing a high quality *library* of reusable code that other researchers may use. One example of such a product is the bioconductor package for the `R` programming language.

This class is about the former, although many concepts will apply to the latter. In general, teaching about the standards required for a general-purpose tool to have impact is beyond the scope of this course.

## 5.2 What language should I learn?

Unfortunately, the answer to this question often devolves into "language war" polemics. "Language $X$ sucks because of $Y$", etc.. These arguments are unhelpful beyond getting lots of retweets or visits to your blog. The reality is that you will probably be best off with some skills from a few languages rather than a mastery of a single one. Here, we take a practical approach and focus on two languages that are popular in "data science" circles. These are Python and R, which we introduce in Chapters 6 and 7. I give some guidance regarding language choice beyond these two in Section 5.4.1 below and in Appendix C.

## 5.3 Where to get help?

Like many tasks related to research, programming can be frustrating. Why does $X$ not work, etc.? Many of you are familiar with online sources of help, such as stack overflow. These sources tend to be moderated Q&A forums where volunteers respond to questions from the community. These resources are invaluable, but are often imperfect. Answers can be old, out-of-date, or even wrong. Often, out-of-date and wrong go together, as the answer refers to an old version of the language/library/etc. being asked about. It can be quite hard for newcomers to tell good from bad answers. It is also important to not get distracted by discussion threads that go down a rabbit hole of pedantry. A semi-reliable indicator of a good answer is a question that generated a lot of conversation resulting in an "accepted" answer with hundreds of votes. Definitely use these sites, but keep your critical hat on while looking at the answer. Don't just copy/paste and pray that your problems have been solved. Rather, try to understand why the solution works. And, if you do get a solution there, make a note of it in your code so that future you knows where it came from!

My personal opinion is that good books are invaluable resources. They are often written by language experts and thus focus on the idiomatic way to do things. In later chapters, I will give citations to good books and, when possible, give you links to free versions of those books when they do not violate copyright.

## 5.4 Basic programming concepts

### 5.4.1 Compiled versus interpreted languages

It is helpful to subdivide programming languages by what happens to the files you write. For many languages, your code is *compiled* from a somewhat human-readable syntax into a machine-readable format that is then executed. For other languages, your code is processed by an *interpreter* and then executed.

Python and R, the languages emphasizes here, are interpreted languages, which is important. First, it means that the speed at which your code runs *may* be slow relative to the same code written in a compiled language. However, it is often faster to write code in interpreted languages. Such languages often have a richer, more readable syntax than compiled languages, and they often have more features built-in to their standard libraries (see 5.4.2 below). You also don't have to wait for your program to compile every time you change something. Compile times can be substantial for languages like C++.

For most of your work, I will wager that the performance of interpreted languages is more than sufficient most of the time. In fact, the internal guts of most interpreted languages are written in high-performance

 

compiled languages like C when performance matters. For example, R's linear algebra support is all written in C, which is completely hidden from the user who only sees a set of R functions. In Chapters 6 and 7, I'll give some hints about what to do if you run into performance bottlenecks.

One nice feature of interpreted languages is that they can be used two ways. You can write all of your code in a file, and then run it. Or, you can use the language's interactive shell environment to enter commands and to exploratory analysis in real time. These shells allow the languages to be parts of notebooks, which are the topic of Chapter 9.

### 5.4.2 "Standard" libraries

The standard library of a language are the features that come with a "plain" installation of the language. In other words, if you install R or Python, by definition you get a certain set of features. Features are added to languages via *libraries*, which may go by synonyms such as packages, modules, or frameworks, depending on the nomenclature traditions of the language.

A big plus of Python and R is that they have rich standard libraries, which is typical of interpreted languages. For example, R has built-in support for high-performance linear algebra operations. The C programming language, which is compiled, has no such support, and one needs to depend on third-party packages adding those features, such as the GNU Scientific Library.

The standard features of a standard library differ from language to language. There are no rules. R has built-in support for graphics, which makes a lot of sense. R is, after all, primarily a language for statistics. In Python, you need to install matplotlib to start making plots. To get the same linear algebra capabilities in Python that you have in R, you need to install numpy.

### 5.4.3 The components of programming languages

We now turn to the anatomical elements of programming languages. While these concepts are close to universal, we have to pick a language to use in order to show concrete examples. I'll use R here, mostly because it has a traditional syntax for writing the code whereas Python is a bit funky at first.

#### 5.4.3.1 Types

A *type* refers to a specific sort of data. For example, integers and numbers with decimal places are different types in post languages. Characters are often another type, etc.. This section outlines some of the more common types.

**5.4.3.1.1 Scalars** Take a single integer value, say `5`. Such single values are referred to as scalar values.

**5.4.3.1.2 Arrays** Arrays, in contrast to scalars, hold multiple values. For example, we may have an array containing the five scalar values `[1, 2, 3, 4, 11]`.

Arrays may have higher dimensions, and may be referred to as matrices.

**5.4.3.1.3 Associative Containers** An *associative container* allows a mapping of *keys* onto *values*. A dictionary is an associative container. The words are the keys and the values are their definitions. There are many forms of associative containers. They keys may or may not be required to be unique. The keys may be stored in some sorted manner or using a method that may seems like a black box. The requirements for the value may vary, too.

**5.4.3.1.4 Strings** A string is an array of characters. Most of this book is strings.

 

**5.4.3.1.5 Regular expressions, or "regexes"** A large chunk of computing tasks involve text processing. Sometimes we are interested in matching an exact phrase. Often, though, we want to do something like find all occurrences of a *pattern* in a string. For example, find all words in a document beginning with a capital letter. These more abstract patterns are found using *regular expressions*, or *regexes* for short. Both `R` and Python support searching strings with regexes. In `R`, look to the `stringr` library. For Python, the relevant module is `re`, which is part of the standard library. Regexes get complicated fast! You'll need to practice a lot at first.

**5.4.3.1.6 Objects** It is common for programming languages to support more complex types, variably referred to as `structs` (structures), `classes`, or some other term meaning a more complex data type than a simple number or character. We will introduce these in later chapters as needed, as the details are language-specific.

#### 5.4.3.2 Variables

Programming languages use variables to store type information and associate it with a name:

```
x <- 5
print(x)
```

```
## [1] 5
```

Idiomatically, R uses this wacko `<-` syntax for variable assignment. In fact, this arrow used to be a button on older computers, which is where R gets it from. Pretty much every other language on the planet uses `=` for assignment. I will admit to never using `<-` in my own R code, which will make me the subject of abuse on Twitter if word got out! There are cases where it matters which assignment method you use, but I've decided not to worry about it. Twitter flame wars aside, this seems like a small point, but it is an example of how there are many nearly equivalent ways of doing the same thing in any language.

#### 5.4.3.3 Keywords

A *keyword* is a fundamental part of a language's syntax. A keyword may not be used as a variable name because it would confuse parsing the code

For example, the following block of `R` code are invalid because I am attempting to use a keyword as a variable name:

```
if = 4
for = 13245
```

#### 5.4.3.4 "Null" and other special values.

We need a way to express the concept of a variable being undefined. These "null" values can represent that a variable is undefined, or has a value that resulted from a domain error in a mathematical calculation. For example, in R, the NULL value is `NA`:

```
a <- NA
```

There is also a representation of infinity and negative infinity:

```
b <- Inf
c <- -Inf
```

Functions exist to check these values. In R,

```
is.na(a)
```

```
## [1] TRUE
```

 

```
is.na(b)
```

```
## [1] FALSE
```

```
is.infinite(b)
```

```
## [1] TRUE
```

```
is.infinite(c)
```

```
## [1] TRUE
```

#### 5.4.3.5 Mathematical operations

We often need to do math on our numeric variables. The syntax is usually intuitive:

```
y <- 2
# addition
z <- x + y
# subtraction
z <- x - y
# multiplication
z <- x * y
# division
z <- x/y
```

The `+`, `-`, etc., symbols are called *operators*.

#### 5.4.3.6 Flow control

For our purposes, computers are simply logic execution boxes. They do what we tell them to, and we can affect how, when, and even if a chunk of code is executed via various "flow control" mechanisms.

We can execute code if and only if a certain condition is met:

```
if (x == 5)
{
    print("Yes, x is indeed 5!")
} else {
    print("No, sorry!")
}
```

```
## [1] "Yes, x is indeed 5!"
```

Languages also provide `for` and `while` loops for flow control:

```
for (i in 1:5)
{
    if (i %% 2 == 0.0)
    {
        print(paste(i,"is even"))
    } else {
        print(paste(i,"is odd"))
    }
}
```

```
## [1] "1 is odd"
```

 

```
## [1] "2 is even"
## [1] "3 is odd"
## [1] "4 is even"
## [1] "5 is odd"
```

The above is equivalent to:

```
i = 1
while (i < 6)
{
    if (i %% 2 == 0.0)
    {
        print(paste(i,"is even"))
    } else {
        print(paste(i,"is odd"))
    }
    i = i + 1
}
```

```
## [1] "1 is odd"
## [1] "2 is even"
## [1] "3 is odd"
## [1] "4 is even"
## [1] "5 is odd"
```

By the way, you just learned that `%%` is R's "modulo" operator, which means that `a %% b` returns the remainder of `a/b`.

The various methods of flow control can be mixed, matched, and intermingled.

#### 5.4.3.7 Functions

A fundamental method for organizing code is to write *functions* to perform specific tasks:

```
addtwo <- function(x, y)
{
    return (x + y)
}

x = addtwo(3,-11)
print(x)
```

```
## [1] -8
```

The way to read the above code is, "There is a function called `addtwo` that takes two arguments, `x` and `y`. The function returns a single value."

Functions are useful because they can make the logic of a program easier to read. The following is pseudocode:

```
filename = "datafile.txt"
z = setup(filename)
if (!is.na(z))
{
    process_data(z)
} else {
    stop("Oops--there is an error!")
}
```

 

#### 5.4.3.8 Comments

All languages support the inclusion of lines of text that are not evaluated as code. These lines are called "comments". Comments are useful to documenting especially tricky logic, or the fact that you changed something at a certain time, or where you got the idea for the code from:

```
# The implementation of foo
# inspired by (link to, say, stack overflow)
foo <- function(x)
{
    # ...
}
```

In R and Python, comments begin with a hash sign (#).

In many languages, include Python and R, it is possible to generate documentation for code directly from the comments.

#### 5.4.3.9 Error handling

Things go wrong when programs run. Languages provide means of handling those errors. Above, we saw the R function `stop`, which is used to signal that an error has occurred, printing the message provided.

Error handling is an important part of "defensive programming", which I discuss more below.

#### 5.4.3.10 Getting help

R and Python have built-in help systems. For example, if you don't know what `is.na` does in the above block, the following command will show you the help page:

```
help(is.na)
```

## 5.5 Programming idioms

### 5.5.1 split/apply/combine

See Wickham (2011) for discussion.

## 5.6 General advice

### 5.6.1 RTFM

**R**ead **t**he **F**ull **M**anual. You'll see this acronym out there a lot, and it is good advice, although it is crude. Basically, make sure you read the documentation for the functions you are using. What does a function do? What guarantees does it make? What side effects are there?

An example of where this matters is described here. A function in the Python standard library resulted in different answers on different operation systems, causing a large number of scientific papers to report incorrect results. The reason is a failure to RTFM. The method used to find the files on the computer is documented as returning them in "an arbitrary order", which should have been an immediate red flag. Thanks to Edwin Solares for pointing this out to me.

 

### 5.6.2 Write functions

In R and Python, you can write your script "top to bottom" as a list of instructions to execute. Doing so is *totally fine*! However, it breaks down the longer your script gets. Almost any top-to-bottom logic can be rewritten as a set of smaller functions. It takes more time, but it forces you to think about what you are doing at each step, it makes error handling easier, and it makes it easier for your colleagues and future you to understand your code six months later.

### 5.6.3 Keep functions short

Ideally, a function does three things:

1. Checks that the arguments passed in are okay. If not, signal some kind of error.
2. Perform a single task given the input arguments. If the task goes badly, signal some kind of error
3. Optionally, return some data back to where the function was called from.

This advice is related to some trendy jargon called the "one responsibility principle".

Related advice is to name your functions clearly after what they are doing! For example, this is unreadable:

```
x <- function(y)
{
}

z <- function(a, b, c)
{
}
```

You need quite a few comments to understand what these functions do!

### 5.6.4 Expect errors and test for them

It is really helpful to be able to catch errors where they happen and report the error. When processing big data sets, you may come across unexpected input like missing data, etc., that your code doesn't handle. With time, you'll learn how to test for these things and automatically write functions that return error messages as early as possible.

When working on shared clusters like the UCI HPC, error checking is paramount! It is really easy for your work to fail "silently" because the error message was suppressed.

The next chapters will have more on this topic.

### 5.6.5 Editing code

A good "programmer's" text editor is essential. Like choosing a programming language, editor choice is prone to pointless online debate. However, there are some objective truths: word processors are not acceptable! You **must** use an editor that saves files in plain text, preferably with Unix newline characters (otherwise your life will be hard on the cluster). The two classics are emacs and vim. I used emacs for about 15 years before pain (google "emacs pinky") caused me to switch to vim, and specifically to neovim.

Other options that people like include, but are not limited to:

- atom
- Microsoft VS code
- nano

A good editor should have the following features:

 

- Syntax highlighting, which means that keywords, variables, etc., have separate colors. This feature greatly improves readability.
- The ability to execute shell commands from within the editor is very handy
- `git` integration is *very* nice to have

 

# Chapter 6

# The Python programming language

## 6.1 Good books

For beginners, I recommend the Software Carpentry lessons (see Chapter 5). McKinney (2017) and VanderPlas (2016) give a good overview of Python for "data science" applications. Ramalho (2015) is an excellent book on intermediate to advanced Python programming.

## 6.2 Strengths and weaknesses

Python is an excellent *general purpose* programming language, as opposed to a *domain specific* language that primarily supports writing code for a limited set of tasks. Mathematica and SAS are examples of commercial, domain-specific languages, although the companies behind them would probably disagree.

Python is generally considered easy to learn, although any such statement is subjective.

On the community side, Python is a welcoming and inclusive community. That doesn't mean that bad ideas won't get a "hard no" response, especially if you were to propose a change to the core language that is not well thought out. What it does mean is that people are pretty good about responding to questions and helping people out, modulo the usual concern with open source software that this sort of support is done entirely on a volunteer basis.

Regarding data-centric programming, Python is the current *lingua franca* of machine learning and artificial intelligence tools. It also supports object-oriented programming idioms, streamlining the design of more complex code bases. (The object-oriented stuff is covered in Ramalho (2015).)

Python has one small and one big weakness, at least as far as we are concerned. The small weakness is that Python code is sensitive to the *white space* in your file, which makes it very different from most other languages. This means that the indentation in your code must be consistent, which actually forces you to write short lines of code that are nicely-formatted. However, it also freaks people out at first. The solution to your spacing problem is solved by your editor:

- Set a `tab` character to equal four spaces in length
- Set the `tab` command to insert spaces instead of a tab.

The combination of these two settings means that your file will display the same way in all editors, which is extremely helpful when multiple people are collaborating. For fun, you should now do a Google search on "silicon valley tabs versus spaces", if you haven't seen it already...

The bigger problem with Python is that a "data frame" type is not as well-developed as its R analog. A data frame is, for all intents and purposes, a representation of a spreadsheet in memory. In other words, we have rectangular data where columns represent variables and rows are observations of those variables. I discuss this point more in 6.15 below.

If you work more with Python, you'll eventually run into another weakness. The standards required to package your software for distribution via official channels such as PyPi are loose at best, making distribution a somewhat frustrating endeavor.

## 6.3 Python 2 vs 3

You must be using Python version 3, preferably 3.6 or later. Python 2.7 is the last release of the 2.x series, and it is slated for end of life January 1, 2020, meaning that no more updates will happen. The major libraries for Python, some of which are discussed below, have already dropped support for 2.7.

Note, installing Python 3 on Apple's `macOS` operating system is best done via `conda`. See Chapter 12.1.

## 6.4 Python files

A file containing Python code ends with the suffix `.py`.

## 6.5 Executing a script

To execute a set of commands in a script:

```
python3 scriptname.py
```

On systems using `conda` (Chapter 12.1) to provide Python 3, the following will work:

```
python scriptname.py
```

## 6.6 Shells

As an interpreted language, Python lets you work in an interactive shell. The commands `python3` or `python` will start that shell. Once the shell is loaded, you can enter the same types of Python commands that you would type into a script. This shell is useful for playing around with ideas to get something working.

A nicer version of the shell is called iPython, which has more features and is the core technology behind Jupyter notebooks (see Chapter 9).

## 6.7 Modules

Python libraries are called modules and are loaded via the `import` command:

```
import numpy
```

The name `numpy` is a *namespace*, and using types in a module requires a reference via the namespace:

```
a = numpy.arange(11)
print(a)
```

```
## [ 0  1  2  3  4  5  6  7  8  9 10]
```

 

To save yourself some typing, you may provide a *namespace alias*. For example, most people will import `numpy` as follows:

```
import numpy as np
a = np.arange(7) # Note the shorter namespace!
print(a)
```

```
## [0 1 2 3 4 5 6]
```

## 6.8 Useful types

The Python language provides a rich set of types for you to work with. This section serves to name some of the more common types. See the official docs and/or online tutorials for more detail.

See Section 5.4.3.1 to remind yourself of basic definitions.

### 6.8.1 None

The NULL type in Python is called `None`:

```
x = None
if x is None:
    print("yep, x is None!")
```

```
## yep, x is None!
```

### 6.8.2 str

Data consisting entirely of characters are represented as strings, or the `str` type:

```
dna = 'AGCT'
type(dna)
```

```
## <class 'str'>
```

String can be manipulated in various ways, see the official docs.

### 6.8.3 Regular expressions

The `re` module provides regular expressions (regexes), which are a powerful way of searching and manipulating strings using patterns. In general, regexes get complicated fast!

### 6.8.4 bytes (what you may want instead of str)

The Python `str` type has a Unicode encoding, which is great for portably representing strings across systems using character sets from different languages. However, it is *not* the classical definition of a string, which is often taken to mean a raw buffer of character data. Such raw buffers are the `bytes` type:

```
dna = b'AGCT'
type(dna)
```

```
## <class 'bytes'>
```

A Google search of "python strings vs bytes" will take you to more on this topic.

 

### 6.8.5 list

The `list` is the array-like type provided by the standard library:

```
x = [1, 2, 3, 4]
type(x)
```

```
## <class 'list'>
```

There are lots of fancy ways to create them:

```
x = [i for i in range(1, 11, 2)]
x
```

```
## [1, 3, 5, 7, 9]
```

You access elements via the `[]` operator. In Python, the index if the first element is zero!

```
x[0]
```

```
## 1
```

Python supports all sorts of fancy indexing:

```
x[-1]
```

```
## 9
```

```
x[::2]
```

```
## [1, 5, 9]
```

```
x[::-1]
```

```
## [9, 7, 5, 3, 1]
```

Some notes:

- The types contained within a list need not all be the same
- The length of a list is given by the `len` function, which works for most containers in Python

### 6.8.6 dict

The `dict` is the basic form of associative container:

```
dna2rna = {'A':'U', 'G':'C', 'C':'G', 'T':'A'}
dna2rna['A']
```

```
## 'U'
```

```
rna = ""
for i in dna.decode('utf-8'):
    rna += dna2rna[i]
rna
```

```
## 'UCGA'
```

Some hints about the above:

- We previously declared `dna` as a `bytes` object, but our `dna2rna` contains `str` types.
- Thus, we need to decode the bytes to a string.

We could do the above entirely using `bytes` types, but it is more complex:

 

```
dna2rna = {b'A':b'U', b'G':b'C', b'C':b'G', b'T':b'A'}
dna2rna[b'A']
```

```
## b'U'
```

```
rna = b''
for i in dna:
    rna += dna2rna[chr(i).encode()]
rna
```

```
## b'UCGA'
```

It is useful to note that the `dict` is what you want when you need a fast lookup table. However, there is no guaranteed order:

```
for key, value in dna2rna.items():
    print(key, value)
```

```
## b'A' b'U'
## b'G' b'C'
## b'C' b'G'
## b'T' b'A'
```

The output order happens to be the same as the input order here, but that will not be generally true.

If the order matters:

```
import collections
ordered_dna2rna = collections.OrderedDict({b'A':b'U', b'G':b'C', b'C':b'G', b'T':b'A'})
for key, value in ordered_dna2rna.items():
    print(key, value)
```

```
## b'A' b'U'
## b'G' b'C'
## b'C' b'G'
## b'T' b'A'
```

### 6.8.7 tuple

A `tuple` is an immutable collection:

```
x = (1, 'abcd')
type(x)
```

```
## <class 'tuple'>
```

The following code will trigger an error, and thus is commented out:

```
# x[0] = 2
```

Tuples are often used for returning multiple things from functions.

## 6.9 Handling files and interaction with the operating system

The `sys` and `os` modules provide functions for talking to the OS. The `glob` module is very handy for finding files based on patterns such as `*.txt`.

 

Do not assume that functions in these modules have identical behavior on different operating systems, or even on different versions of the same OS. For example, `glob.glob` is documented to return the list of files in an "arbitrary order". Thus, if the order of the files matters to your work flow, you need to take the extra step to sort that list yourself according to the desired criteria.

## 6.10 Functions

Functions are defined using the `def` keyword:

```
def add(x, y):
    return x+y

a = 1
b = 2
add(a, b)
```

```
## 3
```

Functions can have default values:

```
def fxn_with_default(a = None):
    return a

fxn_with_default()
fxn_with_default("Hello!")
```

```
## 'Hello!'
```

The first function call doesn't print anything because it returns the NULL type `None`.

## 6.11 Flow control idioms

### 6.11.1 if/else

```
x = 11
if x > 0 and x < 5:
    print ("condition 1")
elif x >= 5 and x < 32:
    print("condition 2")
else:
    print("condition 3")
```

```
## condition 2
```

### 6.11.2 For loops

```
sum = 0
for i in range(10):
    sum += i
print(sum)
```

```
## 45
```

 

```
x = [i for i in range(10)]
sum = 0
for i in x:
    sum += i
print(sum)
```

```
## 45
```

Note that the creation of `x` involved a `for` loop buried inside a *list comprehension*, which is a Python method for creating loops via one-liners.

### 6.11.3 While loops

```
isodd = True
i = 0
while isodd is True:
    if x[i] % 2 == 0.0:
        isodd = False
    print(x[i])
    i += 1
```

```
## 0
```

## 6.12 Error handling

Python has two methods for reporting errors at run time. The first is to "throw" an error object called an `Exception`. The second method is to `assert` that something is true. The latter method can be disabled at runtime by invoking the Python interpreter with the "optimization flag":

```
python3 -O scriptname.py
```

### 6.12.1 Exceptions

When an exception is thrown, the interpreter reports the type of the exception, the exception message, and what file/line number did the throwing.

In most cases, that is all that you need to know! It is pretty obvious when an error occurs.

By way of example, let's use our `add` function from above, but we'll pass in two variables that cannot be added together:

```
try:
    add(1, "bananas")
except:
    print("Hmmm, that didn't work...")
```

```
## Hmmm, that didn't work...
```

We can get more info:

```
try:
    add(1, "bananas")
except Exception as e:
    print(e)
```

 

```
## unsupported operand type(s) for +: 'int' and 'str'
```

### 6.12.2 The `assert` statement

The `assert` statement is a useful way of saying "if this condition does not hold, abandon ship immediately".

The general form is:

```
assert condition, "Condition not met!"
```

For example (the following is unevaluated as it would prevent the book from being generated):

```
x = None
assert x is not None, "fatal error: value is None"
```

### 6.12.3 Which to use?

Well, both! But, if you want your user to be able to gracefully recover from an error, prefer exceptions. In general, prefer exceptions and reserve `assert` for cases that really shouldn't happen. I tend to sprinkle `assert` statements into code that I am trying to debug.

## 6.13 numpy

The **Num**meric **Py**thon library, or `numpy`, is a high performance array and numerical computing library. The name is pronounced "numb pie", but rhyming it with "lumpy" is a common mispronunciation.

`numpy` is implemented in C, and provides high-performance multidimensional arrays and linear algebra functionality to Python. Although it is not part of the standard library, it may as well be.

VanderPlas (2016) is an excellent introduction to `numpy`.

The fundamental type is the `array`:

```
import numpy as np

x = np.array([1, 11, -10, 14])
x
```

```
## array([  1,  11, -10,  14])
```

```
x.dtype
```

```
## dtype('int64')
```

The `dtype` is the numeric type stored by the array. There is support for the full range of numeric types found in C:

```
x = np.array([1, 11, -10, 14], dtype=np.float)
x.dtype
```

```
## dtype('float64')
```

```
x = np.array([1, 11, -10, 14], dtype=np.int32)
x.dtype
```

```
## dtype('int32')
```

```
x = np.array([1, 11, -10, 14], dtype=np.int8)
x.dtype
```

 

```
## dtype('int8')
```

For C/C++ programmers, it is fine to be confused that `np.float` is the C/C++ `double`. A complete list of valid `dtypes` can be found online. (In fact, you can create your own using C or C++!)

The arrays can be multidimensional:

```
x = x.reshape((2,2))
x
```

```
## array([[  1,  11],
##        [-10,  14]], dtype=int8)
```

```
x.shape
```

```
## (2, 2)
```

Access is "C-major" by default, which means rows first, then columns:

```
x[0,:] # row 0
```

```
## array([ 1, 11], dtype=int8)
```

```
x[1,:] # row 1
```

```
## array([-10,  14], dtype=int8)
```

```
x[:,0] # column 0
```

```
## array([  1, -10], dtype=int8)
```

```
x[:,1] # column 1
```

```
## array([11, 14], dtype=int8)
```

There is full support for matrix and vector maths. However, the usual mathematical operators (`+`, `-`, etc.) work element-wise, and this don't give the expected results:

```
x + 2
```

```
## array([[ 3, 13],
##        [-8, 16]], dtype=int8)
```

```
x * np.array([-1, 1], dtype=x.dtype)
```

```
## array([[-1, 11],
##        [10, 14]], dtype=int8)
```

To get the expected results from a vector-by-matrix operation, use the `dot` function:

```
x.dot(np.array([-1, 1], dtype=x.dtype))
```

```
## array([10, 24], dtype=int8)
```

For a full list of `numpy` features, see the home page.

## 6.14 scipy

`numpy` is a part of a larger community of scientific software for Python called `scipy`. In an ideal world, the former would provide the array types and some basic operations on them, and the latter would contain all of the numerical code. The reality is that there is some feature overlap with both relying on the `numpy.array`

 

as a fundamental type. `scipy` contains code for all sorts of statistical calculations, numerical optimization methods, etc.. In many ways, `scipy` adds a lot of the functionality found by default in R.

## 6.15 Pandas (not the animal...)

The pandas package provides a data frame object to Python plus many methods of operating on such objects. See McKinney (2017) for a good overview of the library.

`pandas` provides functions for getting data to/from files as well as to/from databases (see Chapter 11). For example, let's read in the `mtcars` data that is commonly use for R examples:

```
import pandas as pd
mtcars = pd.read_csv('data/mtcars.txt', sep='\t', index_col=0)
type(mtcars)
```

```
## <class 'pandas.core.frame.DataFrame'>
```

```
mtcars.head()
```

```
##                    mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  carb
## Mazda RX4         21.0    6  160.0  110  3.90  2.620  16.46   0   1     4     4
## Mazda RX4 Wag     21.0    6  160.0  110  3.90  2.875  17.02   0   1     4     4
## Datsun 710        22.8    4  108.0   93  3.85  2.320  18.61   1   1     4     1
## Hornet 4 Drive    21.4    6  258.0  110  3.08  3.215  19.44   1   0     3     1
## Hornet Sportabout 18.7    8  360.0  175  3.15  3.440  17.02   0   0     3     2
```

I like to describe the `pd.DataFrame` object as somewhere in between a spreadsheet and a database. `pandas` relies quite a bit on the idea that one of the columns is an index column, much like how databases work. This behavior differs from the R `data.frame` object (see Chapter 7).

Internally, the columns of a `pd.DataFrame` are `numpy` arrays. Thus, they are high performance but do not have a universal notion of a NULL value at the time of this writing. For the latest, do a search for "panda missing data".

I personally use `pandas` routinely in my research. However, I must acknowledge that it is an idiosyncratic package. While it is a powerful tool, its interface is quirky, with the same or similar options having different names in different functions, etc.. Part of the problem is the usual one of open-source software: `pandas` was developed at a company for the analysis of time series financial data. After open-sourcing it, the package quickly became very popular relative to the number of people working on it. The result is a package trying to catch up and evolve into a more polished state. The project leader, Wes McKinney, even has a blog post on what he doesn't like about `pandas`.

Acknowledging the issues described above, `pandas` provides powerful methods for analyzing data frames. There is a lot of attention to detail in the `pandas` code in terms of performance, meaning that many operations are very efficient.

There are methods to group data and summarise subsets:

```
# Group by automatic or manual transmission and number of gears,
# getting the mean of remaining columns
g = mtcars.groupby(['am','gear']).mean()
g
```

```
##                mpg       cyl      disp          hp      drat      wt    qsec    vs      carb
## am gear
## 0  3     16.106667  7.466667  326.3000  176.133333  3.132667  3.8926  17.692  0.20  2.666667
##    4     21.050000  5.000000  155.6750  100.750000  3.862500  3.3050  20.025  1.00  3.000000
```

```
## 1  4     26.275000  4.500000  106.6875   83.875000  4.133750  2.2725  18.435  0.75  2.000000
##    5     21.380000  6.000000  202.4800  195.600000  3.916000  2.6326  15.640  0.20  4.400000
```

There are also database-like methods for joining data frames together.

## 6.16 Improving performance

Most of the time, Python code written using the Python standard library, `numpy`, and `pandas` will be "fast enough". If you find yourself in a situation where you have a performance bottleneck, you can look at the following tools that may help.

Note that none of these tools promise speedups! In fact, you will often be wasting your time unless you *know* where your performance problem comes from in your code! The only way to do that is to *profile* your code, and how to do that is BTS. For all of these tools, the best use is targeted, meaning that you want to only replace your bottleneck with the (hopefully) more efficient version.

### 6.16.1 Cython

Cython is a compiler that translates a Python-like syntax into C code and compiles it. There is special support for `numpy` arrays and you may make "wrappers" to functions provided by C or C++ libraries, although the C++ support is limited.

Fun tip: CythonGSL is a fairly complete interface to the GNU Scientific Library.

### 6.16.2 numba

numba attempts to compile Python code into machine code. The basic premise is that you can take a function that is slowing down your code and get a faster compiled version. The documentation suggests that you'll get the best results for code doing numeric operations.

### 6.16.3 The C API

Python is written in C, and has a full C library that you can access. Using this library, you can write Python modules in a mixture of C and Python. Generally speaking, this is *quite advanced*.

### 6.16.4 pybind11

pybind11 is a C++ library allowing you to write Python modules in idiomatic C++. The big feature is that it supports modern C++ standards, meaning C++11 or later. I use `pybind11` for a lot of my research code. It is a fantastic product. Like the C API, this is an advanced topic.

 

# Chapter 7

# The R programming language

This chapter does not cover the task of programming using R in as much detail as we did for Python in Chapter 6. The difference does not imply a criticism of R. Rather, it reflects some practical realities of where my time is best spent and R will take priority over Python in Chapters 10 and 11.

## 7.1 Good books

Wickham and Grolemund (2017) is an essential book covering the `tidyverse` (see 7.4).

## 7.2 Strengths and weaknesses

R is a domain-specific language for statistical analysis. A major strength of R is a pretty full-featured standard library, usually referred to as `base` R (7.3).

Like Python, a huge array of helpful libraries are available. Also like Python, it isn't always easy to discover the one that you need.

Unlike Python, the guidelines for creating R packages for distribution are highly detailed and strictly adhered to, creating a high average quality of packages installed via CRAN. Unfortunately, violations of these strict rules can be reported in ways that are often viewed as unwelcoming and non-inclusive. Certain persons on the R mailing lists are infamous.

Regarding the language itself, most of the weaknesses are found in `base`. The development of the core language is highly conservative, meaning that some corners of the language that are klunky have been so for a long time and will likely be so forever. You either get used to the oddities or decide to be in a constant state of vexation. In many cases, there are libraries on CRAN to address some of these issues.

## 7.3 base

The standard library excels at the manipulation and processing of rectangular data, meaning "spreadsheet-like" data arranged into columns and rows. The reason for this strength is that a date frame type is a core language feature.

R is an excellent platform for all things linear modeling. There are entire libraries and books on the topic. In fact, searching Amazon will show you that a rich library exists that covers all sorts of common and esoteric statistical methods and how to do them in R.

R has built-in support for matrix and vector types as well as linear algebraic operations. These operations, and many of the low-level linear regression functions, are implemented in C and are very well-tested.

The standard library, or `base` R (7.3) has good support for plotting, which we discuss more in Chapter 10.

Many basic concepts of R were introduced in 5.4.3, so we can focus here on other useful features.

R has five fundamental data structures: vectors, lists, matrices, arrays, and data frames. It is quite easy to get some or all of them confused with one another. If you need to get into the details of these types for your work, then you need to read about them, and there are many online tutorials to help you out.

For our purposes, the `data.frame` is where most of the action is:

```
a = 1:4
b = c(TRUE, FALSE, TRUE, FALSE)
c = c("lemming", "potato", "badger", "tomato")

tasty = data.frame(name=c, isanimal=b, tastiness=a)
tasty
```

```
##      name isanimal tastiness
## 1 lemming     TRUE         1
## 2  potato    FALSE         2
## 3  badger     TRUE         3
## 4  tomato    FALSE         4
```

The `data.frame` is our analog of an in-memory spreadsheet. We can do group and summarise operations via the `aggregate` function. The next code chunk gives us the mean tastiness for each value of `isanimal`:

```
mean_by_isanimal = aggregate(x=tasty[c('tastiness')],
                             by=tasty[c("isanimal")],
                             FUN=mean)
mean_by_isanimal
```

```
##   isanimal tastiness
## 1    FALSE         3
## 2     TRUE         2
```

The built in `aggregate` works, but you may find yourself quickly pushing its limits. I will admit my bias towards doing these sorts of operations with the `dplyr` library in the so-called `tidyverse`. The main reason is for improved performance on large data sets. The second reason is that you get a toolkit for processing databases for free (see Chapter 11).

## 7.4 tidyverse

The `tidyverse` is a set of R libraries intended to provide a modernized interface to tasks related to the analysis and visualization of data. It is a collection of many libraries with contributions from many people. Wickham and Grolemund (2017) is an excellent book on these libraries. A free version is available online from the authors, but it is not identical to the published version. The chapters are in a different order, but most of the content seems to be there.

The `tidyverse` provides plotting capabilities via the very popular `ggplot2` library, which is covered in 10.3.2.3. Other key libraries include `stringr` and `readr`, which you should definitely learn about from the book mentioned above!

The `dplyr` package is an invaluable tool for manipulated `data.frame` and `tibble` objects. (The latter is an augmented `data.frame`–see the Wickham and Grolemund (2017) book.)

 

### 7.4.1 dplyr

The dplyr library provides a high level grammar for data frame manipulation. Its interface relies heavily on a new "pipe" operator, `%>%`. This operator is analogous to the pipe operator from the Unix command line (`|`). The origin of the R pipe is the magrittr library.

Let's just dive into `dplyr` and redo the analysis from the previous section:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
mean_tastiness = tasty %>%
    group_by(isanimal) %>%
    summarise(mt = mean(tastiness))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
mean_tastiness
```

```
## # A tibble: 2 x 2
##   isanimal    mt
##   <lgl>    <dbl>
## 1 FALSE        3
## 2 TRUE         2
```

In my opinion, the `dplyr` version is much more readable. I find the "piping" of commands (often called *verbs*) rather elegant. There is a lot you can do with `dplyr` for various "data wrangling" tasks, and Wickham and Grolemund (2017) covers many of them. Further, `dpylr` is quite *efficient*, as most of its core operations are implements in C++ using `Rcpp` (see 7.5).

A common criticism of the pipe-based syntax is that it is harder to find where errors occur. I find this to be a non-issue. When you do run into problems, just "unpipe" things to debug:

```
g = group_by(tasty, isanimal)
s = summarise(g, mt = mean(tastiness))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
s
```

```
## # A tibble: 2 x 2
##   isanimal    mt
##   <lgl>    <dbl>
## 1 FALSE        3
## 2 TRUE         2
```

It is less work to semi-unpipe things when debugging:

```
g = tasty %>% group_by(isanimal)
s = g %>% summarise(mt = mean(tastiness))
```

 

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
s
```

```
## # A tibble: 2 x 2
##   isanimal    mt
##   <lgl>    <dbl>
## 1 FALSE        3
## 2 TRUE         2
```

These latter two blocks of code are a bit less efficient because they create more *temporary objects* in memory.

## 7.5 Improving performance

As with Python, most R code will be fast enough most of the time. When that is not the case, Rcpp is a high-quality library that allows you to write R functions and objects in modern C++. Rcpp is the most widely depended-upon package in the R software ecosystem.

The official way to get help with Rcpp is via the mailing list. Expect a rebuke if you cross-post to Stack Overflow or other such places.

 

# Chapter 8

# git and GitHub (and GitLab, and BitBucket. . . )

Writing code is an iterative process full of false starts, corrections, and more corrections. A common experience is to end up with end up with a directory with a list of files looking something like this:

```
do_analysis.sh
do_analysis_2.sh
do_analysis_fixed.sh
do_analysis.py
plot.R
plot_bad.R
plot_fixed.R
```

This is a world of hurt. You cannot trust the file names. You probably cannot trust the modification dates–are you *really* sure you didn't `rsync` them from another machine, possibly forgetting to preserve the time stamps in the process? If you learn one habit from this class, it will be to start every new project with:

```
mkdir new_project_name
cd new_project_name
git init
```

git is a distributed version control system for code. Before describing it in more detail, we should define code. Code is *any plain text file* related to your research. That includes source code for scripts and programs, `markdown` files, raw data from the field, LaTeX files, etc..

Other types of files may also be okay, within reason. Image files resulting from plots are fair game. Modest-sized compressed files of data may also be okay–the down side here is that if the files change then you cannot see exactly *what* the change was. You would be limited to seeing *who* made the change and *when*.

`git` allows you to do the following:

- Retain a complete history of all edits to your files. This is done via a *commit*, which is the act of saving a change in the *repository* for your project.
- Revert changes that you made to a file. For example, you just made a bunch of changes that turned out to be a bad idea. You can ask `git` to simply revert the file to the last committed version. Or, you discover that you just committed changes by mistake. You can ask `git` to revert to an earlier commit.
- Make and commit changes to the same sets of files *separately* in different *branches* of your project. For example, you may have two different ideas for how to do a plot. You could work on them in separate

branches, eventually deleting the branch that didn't work, only retaining the changes in the branch that you want to keep.
- Share code associated with your publications via uploading it to a website like GitHub.

`git` lets you do a whole lot more, too, but this list is enough for now. And once you get your code online, you have a whole new world of possibilities (see 8.5 below).

Tools like `git` are essential for PhD students now. Too much of your day to day work involving your data involves writing bits of code to reformat or plot those data, and you need an organized way to manage that code. Journals are also likely to require that you deposit the code for your analyses and your plots somewhere, and `GitHub` is one possible place. Data Dryad and figshare are possibilities, too, but there is good reason to be skeptical of their long-term availability. If a journal requires you to upload your code to `Data Dryad`, then my suggestion will be to make a *release* of your code on `GitHub`, download that release for yourself, then upload it.

## 8.1 Basic operations

There is a lot to `git`. This section gives something close to the minimal set of operations you need to know about. Fortunately, `git` is so widely used that there exists an abundance of excellent tutorials online. These two are particularly good, especially the second:

- From the main git page
- From Atlassian

### 8.1.1 Initialize a repository

When starting a new project, do the following:

```
mkdir new_project_name
cd new_project_name
git init
```

The result is a new, empty repository. It contains a single *branch*. A *branch* is a history of changes to the project. For most of you, the branch name is `master`. Branches are very important and are discussed below in 8.2.

**NOTE:** `git`/`GitHub` are in the process of renaming the default branch from `master` to `main`. We will deal with this change in class.

### 8.1.2 Clone an existing repository

If your repository exists elsewhere, you may make a copy of it locally. This action is called *cloning* a repository.

To clone a repo from another location on the same system:

```
git clone /path/to/repo
```

To clone a repo from another machine via `ssh`:

```
git clone user@machine:/path/to/repo
```

To clone from a repo hosted on `GitHub` whose author is `account` (this could be your own account):

```
git clone https://github.com/account/reponame
```

If you have enabled two-factor authentication with `GitHub` and want to clone your own repo, you must exchange `ssh` keys. More on this below in 8.5.1.1.

 

### 8.1.3 Add a file

If you want to add a new file to a repo, or *stage* changes made to a file, then use the following command:

```
git add filename
```

To undo a `git add`:

```
git reset filename
```

### 8.1.4 Commit changes

To commit changes to a file:

```
git commit filename -m "commit message"
```

Now, the changes are recorded in the repository history. The commit message is entered into the *log*, which is a history of the commit messages. These records are for *you* and so should be understandable down the road. It is tempting to write terse messages just to get the task over with, but you may regret that later.

To commit all changed files at once:

```
git commit -am "commit message"
```

### 8.1.5 The interaction between add and commit

You may commit a file without having added it. You may commit all changed files at once via the `commit -am` method. To commit a set of changed files with one commit message, `add` those files and then say `git commit -m "commit message"`. Imagine you've changed four files, but you want a different commit message for three of them:

```
git add file1 file2 file3
git commmit -n "first commit message"
git commit file4 -m "second commit message"
```

### 8.1.6 Undoing changes to a file

You will often find yourself in the position where you have made changes to a file, only to realize that they are a bad idea for one reason or another. With `git`, you can do much better than your editor's `undo` features. To revert a single file back to its last committed version:

```
git checkout filename
```

If you have already staged the file via `git add`, you need to undo the add before `checkout`.

If you have changed a whole bunch of files and want to undo it all, there is a nuclear option to revert them all to their last committed versions:

```
git stash
```

The `stash` command is *reversible*, which is kind of amazing. We don't discuss it here, as the details get complicated, but you can find more info online.

### 8.1.7 Pushing changes to an "upstream" repository

If your repository is a clone from elsewhere, you'll eventually want to send you committed changes back to "home base":

 

```
git push origin main
```

More generally:

```
git push origin branchname
```

### 8.1.8 Perusing the history

To see the history of the current branch:

```
git log
```

The most recent log entries for this book currently look like this:

```
commit 11c7f2f30d7cafea516adece6365afec9ac18a8f (HEAD -> git)
Author: molpopgen <krthornt@uci.edu>
Date:   Sat Nov 9 09:27:28 2019 -0800

    many basic git commands

commit 3dfca9f9bc18e094017caacdebb82ed532e17d7f (origin/git)
Author: molpopgen <krthornt@uci.edu>
Date:   Fri Nov 8 17:52:00 2019 -0800

    fix typo

commit 00b974cb7ed4f13802aef203babf768dd3fa588d
Author: molpopgen <krthornt@uci.edu>
Date:   Fri Nov 8 17:41:42 2019 -0800

    Start writing text on git

commit cbaaf1229869485350f4d4bd52e985d6a247fbe9
Author: molpopgen <krthornt@uci.edu>
Date:   Fri Nov 8 16:28:25 2019 -0800

    Add figure showing work flow with branches.

commit 63c72c8029c22569545c76783da6ec8647b83b38
Author: molpopgen <krthornt@uci.edu>
Date:   Fri Nov 8 15:54:36 2019 -0800

    Add Makefile rule for tikz-based figures outputting to pdf.

commit fd6cb5a001e341536ff2f99c6e4c0a12f423b2a7 (origin/master, master)
```

The history contains the following:

- A `hash` for each commit, such as `00b974cb7ed4f13802aef203babf768dd3fa588d` This is the unique identifier for the commit. You may use these value to force a branch back to any stage in its history (see D).
- The author of the commit. This is important for collaboration.
- The date and time
- The commit message

 

- At the end of some of the commit hashes, text in parentheses refers to branch names. I don't want to go into this too much here, but we'll discuss the first couple entires. The entry `(HEAD -> git)` refers to a commit made locally on a branch called `git`. The next entry, `(origin/git)` refers to a commit on the `git` branch that has been pushed back to the remote origin, which in this case is `GitHub`. Thus, this list shows that there are commits that haven't been sent back to home base.

## 8.2 Working with branches

Imagine that you have the following situation;

- You have a working pipeline that consists of a mix of Python and shell scripts. The shell scripts submit jobs to the cluster and execute the Python scripts. A set of R scripts exist that do some downstream post-processing and maybe make a plot or three.
- A reviewer makes some suggestions that may or may not need to be implemented. Checking into this requires modifications to most of the files in your pipeline.

What to do? You could make all the edits and commits to your `main` branch, but this is not ideal for a few reasons. What if the reviewer's suggestion doesn't pan out? If that's the case, then you need to manually reset your main branch to the last commit corresponding to your submitted paper. It may be tough to find that commit in practice (but see 8.3 for some strategies).

Another strategy would be to make a whole bunch of edits and then `git stash` them if they don't work out. This approach could work, but may not be ideal if many changes are needed. (When a lot of changes are required, it is handy to have a lot of commits. In case of problems, you can later find the commit causing the problem.)

A robust solution is to simply start a new history for these commits. It version control jargon, we will start a new *branch* of our code. This branch will contain the commit history from a parent branch (often `main`), but new commits will not be added to `main`'s history, but rather to that of our branch. We can commit at will to this new branch. If the changes are good, we can *merge* them back into `main`. If not, we can simply delete the entire new branch and literally nothing ever happened to main.

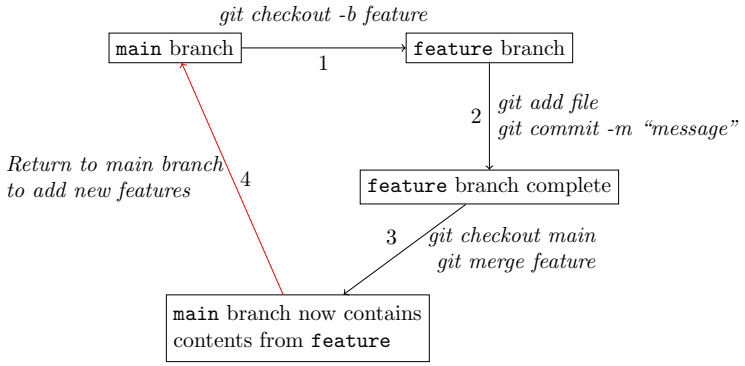


Figure 8.1: Branches with git. Starting with the 'main' branch in the upper left, we create a new branch called 'feature' (1). While working on 'feature', we add files and commit the changes (2). When finished with the work, we switch back to the 'main' branch and merge 'feature' into 'main' (3). The end result is that the 'main' branch is updated to contain the changes from 'feature' (4).

Figure 8.1 shows a schematic of working with branches. In general, you probably don't want to work directly on your `main` branch most of the time (see 8.7). Branches have so many advantages that you should develop the habit of using them whenever possible. The fact that you are not interfering with the `main` history of the

 

project is invaluable. Being able to just delete a bad idea or something that just didn't work out is quite useful.

When you do need to delete a branch:

```
# Go back to main
git checkout main
git branch -d branchname
```

Occasionally, `git` may complain about deleting a branch with changes that haven't been merged. If so, and you know you really do want to delete, use `-D` in place of `-d` in the above commands.

### 8.2.1 Creating branches

When working locally, creating a branch is a two-step process:

```
# Create the branch:
git branch branchname
# Check the  branch out:
git checkout branchname
```

When you create a branch, the new branches history starts off from whatever branch you were just on. For simplicity, we'll assume that that branch was `main`. Thus, and new edits and commits are *on top* of `main`'s history.

Figure 8.1 shows a one-liner, `git checkout -b branchname`. This shorter command says, "if `branchname` does not exist *locally*, create it and check it out". (If it already exists, the command fails and tells you so.) It is easy to get into the habit of always using `checkout -b`, which will eventually cause you to create a local branch with the same name as a remote branch. At this point, your local copy is probably the same history as your remote, and you cannot push it to the remote location.

As long as you haven't made any changes to this local branch, you can execute the following command to make your local branch the same as the remote:

```
git reset --hard origin/branchname
```

**Be very careful with branch resets!!!** This command forcibly overwrites the local history with that of the remote.

If you have accidentally created the same branch name in two different places and have work in each that you want to keep, your life is harder. You may try to merge your local and remote branches via `git pull origin branchname`. However, that may not work out, which brings us to the next section.

### 8.2.2 merge conflicts

I have painted a picture of branches consisting entirely of rainbows and unicorns. There is, however, one situation where branches can get a bit ugly. Considering the following case:

- You are working on two non-`main` branches of your project.
- You merged branch `A`.
- You want to merge branch `B`, which has edited a few of the same files that `A` did.

A lot of the time, this "just works"–`git` can be magical like that. Other times, the edited lines overlap the same lines of the file or lines that are close enough to each other that the intent is ambiguous. In this case, you have a dreaded "merge conflict" on your hands. The merge will be paused and you have to manually edit the conflicted file, add it, and commit it, before the merge can complete. Some good info on how to do this can be found here.

 

A tautological solution to merge conflicts is to not allow them to happen. Just don't mess with the same files from multiple branches. Sometimes you just can't avoid doing so, especially if you are collaborating with someone on a repository.

## 8.3 "Tagging" versions of your code

If we return to the motivating example of using branches, we realize that we haven't covered something important. You just submitted your paper, and the plots are based on a certain set of scripts, and there is a commit in the log. It is a **very good idea** to assign a version number to your code now! It `git` jargon, this is a `tag`:

```
git tag -a first_submission -m "Code used for 1st submission of paper, yay!"
```

Now, the commit hash associated with the last commit is tied to the label `first_submission`. And yes, you can have "version numbers" that aren't numbers, which is pretty great.

You can do a lot with tags, and the docs covers this feature nicely.

## 8.4 The fugitive plugin for vim

The `git` commands shown above are what you would enter into a terminal. It is okay to have two terminal windows open in the same directory–one for your editor, and another to run `git` commands. (Even better would be a terminal multiplexer such as `tmux` or `screen`.) However, it is possible to use `git` directly from within your editor.

This section describes one such editor integration, which is Tim Pope's fugitive plugin for `vim` and `neovim`.

The rest of this section will make no sense if you don't use these editors, so skip if you need to.

`fugitive` provides a series of `normal` mode commands to execute `git` operations. It provides so many commands that you have to go read the README and watch the screencasts to learn about it. What follows is a tiny bit of what it can do. For the following shell command:

```
git add file
git commit -m "made some changes"
```

If `file` is your current buffer, you may use the following `normal` mode commands instead:

```
:Gwrite
:Gcommit
```

The first command does the `git add` part. The second opens a new `vim` buffer where you may type in your commit message. A really nice feature here is that the entire `vim` completion list is available to you, so you can probably auto-complete many of the words in your commit message. Finally, you save your commit message via:

```
:wq
```

Now, you are back in your edit window.

## 8.5 GitHub

GitHub is a web service providing an online location to host `git` repositories. As of mid-2018, it is owned by Microsoft. `GitHub` is the most popular service in this class. Alternatives are GitLab and BitBucket. The latter is owned by Atlassian, and the former is a smaller startup venture. `GitLab` is notable as the host of the `GNOME` desktop for Linux, which moved there from `GitHub` after the Microsoft takeover.

 

In terms of concepts, we will consider these three services as interchangeable. We will only discuss the specific details of the `GitHub` interface, however.

These services allow:

- Code sharing. You can put your `git` repos online using them.
- Collaboration. Groups can push changes to these online repos, discuss them, and merge or reject proposed changes.
- Community organization. Groups of user accounts can be members of Organization accounts. This is a good thing for individual labs to do.
- Host web pages associated with projects. You can even blog!
- A means of getting help and reporting problems with software by submitting "issues" to projects.

You may create repositories at these sites. Once you do, you will get instructions for what to do if you already have a repo locally that you want to push to this new "remote" location.

Your repos may be `private` or `public`. The number of `private` repos depends on the service and the type of account. I consider `private` repos to be of little use for code and moderately useful for papers prior to uploading the preprint and for grants.

### 8.5.1 Setting up accounts

This step is quite straightforward–simply go to the website and follow the instructions.

#### 8.5.1.1 Two factor authentication

I strongly recommend setting up two-factor authentication for your account. There are many ways to do this for GitHub. I personally use a YubiKey, which I also use for my UCI two-factor as it is superior to what UCI recommends, although UCI makes it a bit tricky to set them up. (Yes, they cost about fifty bucks, but you are already paying monthly for your phone's data plan to use the UCI solution.)

Once you set up two-factor, `GitHub` requires you to do `ssh` key exchange for each local machine that you want to use to push commits from. This is a bit tedious, but is a one time setup. I recommend against password-free keys.

### 8.5.2 Issue tracking

One of the best `GitHub` features is issue tracking. By default, active repositories have an `Issues` button allowing you to submit a new issue. Issues are often bug reports and/or feature requests or questions about someone else's project. They are also a great way for you to generate reminders for yourself when a project is in progress. For example, you could submit an issue with the text, "Need to confirm the calculation of X".

When an issue is submitted, it gets a number. Let's assume that our hypothetical issue is assigned number `1`. When you commit a change, you may refer to an issue:

```
git commit analysis.R -m "Added code to verify calculation of X.  Closes #1."
```

The `Closes #1` bit is some magic. First, when this commit is pushed back to `GitHub`, the issue's page will show a link to the commit. Second, if the commit is merged into the `main` branch, then issue number one will automatically *close*, meaning that it is resolved. There are other words you can use to do the same thing, and it is also possible to refer to issues in *other* repositories. You can discover these features online.

### 8.5.3 Pull requests

A *pull request*, or `PR`, is a special way to treat a branch on `GitHub`. A `PR` represents a request to merge changes from a branch into `main`. The `PR` may be accepted and merged or rejected and closed. Like issues,

 

they are assigned a number and you may refer to that number in commit messages. PRs are an important tool for collaboration, and I suggest you read about them.

There is a lot of nuance to effectively using this GitHub feature. One thing to be aware of is that you may want to couple them with issues. For example, you may have a branch called fix_issue_7. You commit a lot of changes to fix this issue. If you merge it via a PR, you may want to edit the commit message to be Closes #7, and then the issue will auto-close.

## 8.6 Common misconceptions about GitHub

I hear several concerns from academics about using GitHub. I believe most of these to be non-issues, and I address them here.

### 8.6.1 GitHub could go away one day

GitHub is a corporate interest. As such, it may close one day. Prior to being purchased by Microsoft, this was a valid concern. After the purchase, I think there is little cause to worry. Even if it does go away one day, it will not be the end of the world for two reasons. First, you will probably just migrate to another service. Two, you already have local copies of your repos, right? (See 8.7).

### 8.6.2 Someone may steal my ideas

This section heading is the most extreme variant of a reason *not* to share code. Another includes not wanting to put things out there that aren't ready. These concerns hold no water. If your lab "does" computational work for its bread-and-butter research, then it should have an "organization" account through one of these services. If you or your lab insists on private repositories, then you may have to pay for them once you reach a certain number. (GitHub has a generous policy here for educators and researchers.)

If you are concerned about releasing code "before it is ready", then there is a misunderstanding at play. Code generated by publicly funded research *should* be open source and open source code benefits from a "release early, release often" approach. The single best way to find bugs and limitations in your code is to have someone else try to use it. The reality is that it will be hard for other to discover your code–unless you publicize it broadly, it is just one of millions of GitHub repos on there. But it may be very useful for your lab mates to try it if they are working on projects where it may help. You may help out on other papers from the lab and you may discover some bugs. Both are good outcomes.

Finally, there is the "someone may steal my ideas" meme. This assumes that someone cares enough, and is malevolent enough, to do so. In most cases, this concern is dealt with via some tough love: **no one cares enough about what you are doing to stalk your GitHub commits**. Think about it for a second. A repo for a project in progress is likely undocumented and the commits are highly technical in nature and the entire work flow likely depends on inputs that are not on GitHub anyways because they are too big. You would need to be working in an extremely competitive area full of bad actors for this to be a likely outcome of using GitHub. Yes, it probably has happened, but I really doubt that it is common.

The solution to all of these concerns, to the extent that they are relevant, is simple: push to a private repo and then make it public *immediately* upon submission to a journal (or upload to a preprint server). The journals that I edit and review for may treat the lack of an available repo as grounds for *editorial rejection without review*, and I expect this standard to spread to other journals.

### 8.6.3 I don't want to be responsible for maintaining this code

This concern reflects a fundamental misunderstanding. Making the code behind your research public is *not* the same as writing a general software package for doing the analyses. The point of sharing code is to document the work flow. In other words, it is fulfilling the ethical duty to demonstrate that the research is

 

reproducible, at least in the sense that the scripts used are available. If you want to indicate that you have no intention of updating the code, archive the repository (see 8.7).

## 8.7 Git and GitHub "best practices"

### 8.7.1 Try not to work on your main branch

Treat your `main` branch as a special place. Ideally, it represents code that is known to work. Use branches for making changes.

### 8.7.2 Use issue tracking

I consider liberal use of issue tracking (8.5.2) as a best practice for tracking the progress of a project. It is very nice to be able to have the issue on a website, possibly with a bunch of comments about it, and then to *know exactly* the commit or set of commits that address the issue. Yes, you could use note-taking programs to manage `TODO` lists for your projects, but they won't have this deeper relationship with your code.

My lab writes most of its papers using LaTeX, and we use `GitHub` during the process. We use issue tracking here, too, to discuss the evolving manuscript, etc..

### 8.7.3 Document your repo

Make sure that your repo has a `README.md` file (`README.rst` is fine, too, for Python projects.)

The contents of this file are up to you. If you do not intend to maintain the code further, then say so here. Otherwise, you may want to point out any required dependencies, etc., needed to use the code.

If the code is a "one off" for a specific project and was only tested on a specific type of system, then it is a good idea to say so. Such details often matter.

### 8.7.4 Add a license to your repo

It is very important that your code contains terms for reuse, which typically take the form of a license. An open source license should be preferred for code resulting from publicly funded research. The specific license that you choose needs to result from a conversation between yourself, your advisors, and your institution. I cannot give specific advice here.

The license for a repository should be in a file called `LICENSE` and/or `COPYING`.

If you are worried about people "stealing" ideas from your repo, you should be aware that there is really no such thing, legally, if your code is out there in an unlicensed form, at least in the US.

### 8.7.5 Document your code if appropriate

If you intend for other to use your code, it is helpful to document it. Use the idiomatic methods for your language(s) of choice to do so.

### 8.7.6 Keep backups of your repos

It is not a bad idea to have local copies of your online repos. For many years, I never pushed straight to `GitHub`. Rather, I cloned from, and pushed to, a local repository. That local repository is what got pushed to `GitHub`. I am less rigid about this now, but I still like the idea in principle.

 

### 8.7.7 Learn to clone between machines

This is a corollary of the above. If you are writing code on your laptop, but intend to run it on your cluster, then the following may make sense:

1. Create the repo on the cluster.
2. Set `GitHub` as the remote origin of the cluster repo.
3. Clone from the cluster to your laptop.
4. Work on your laptop, pushing from laptop to cluster and cluster to `GitHub`.

Now, you have three copies of your work!

### 8.7.8 Do not `rsync` repos around

`rsync` seems like a good way to copy repos from place to place. But it actually doesn't work that well. You will probably find that you cannot make updates to the `rsync`d repo. The private files in a repo have very specific permission requirements, and you need to use esoteric `rsync` flags to preserve them.

I have not tested these flags in a while, but this is from an old script that I used to backup `git` repos using `rsync` that was confirmed to work:

```
rsync -avrRp --update --delete --progress --chmod=Fa-w
```

The correct approach is to clone the repo and then pull all remote branches with

```
git pull --all
```

### 8.7.9 Archive old projects

Code has a life span. For example, the `R` scripts used to make plots for a paper may not work in a year. Tools like `dplyr` and `ggplot` change and a function that you used may no longer exist or how you use it has changed. More generally, a lot of research code exists for a single purpose. When that purpose has been realized, it may be time to "end of life" the code. `GitHub` (and the other services, too) allow you to *archive* a repository. Once archived, a repo is marked as "read only" and issues, pull requests, etc., are no longer allowed. In other words, the repo is only there as an archival document. This is a great end for scripts related to single papers, as it signals you don't intend to make sure that it continues working into the future, but it still meets your obligation of making it available for reproducibility purposes.

 

# Chapter 9

# "Lab notebooks" for computational research: Rstudio, Jupyter, JupyterLab

For lab and field scientists, a laboratory notebook is the fundamental unit of research reporting. If your lab is funded by the NIH, you are required to give you notebook to your PI when you leave. Your PI must keep that notebook for several years. Current guidelines require that records be kept for two years after the grant has closed, but the notes may be needed by your lab for much longer.

A standard lab notebook is relatively straightforward to keep, although it does require some diligence. What do you do for your computational work? How do you go from folders of scripts and output files to something showing an arrangement of your thoughts and results into some sort of "story"?

A partial answer to these questions is to use notebooks as a form of electronic lab notebook. Essentially, a notebook is a means of mixing text, code, and graphics. Isn't that a paper? Yes, although papers usually omit the code. Notebooks are primarily designed for the generation of dynamic reports, meaning that each time the data changes, you can push some button or execute a command, and the code to process the data and make the figures does its thing and the report is updated.

In a highly idealized world, we would write our papers using notebooks. By doing so, we would be stating that all figures, tables, and numbers in the text are based on the latest data files and processing code. For many disciplines, generating manuscripts this way is not feasible. Imagine that Figure 6 if your paper requires 300 CPU hours on your institution's cluster and a few terabytes of space to hold the intermediate results, which are then processed by a series of `R` and/or Python scripts. It is clearly not possible to hit the `knit` button in `R Studio` from your laptop and get the latest results into your manuscript.

If you are interested in notebooks and reproducible science, then I highly recommend following Karl Broman's blog and to also follow him on Twitter.

## 9.1 R Studio (R markdown more generally)

R Studio is a graphical interface to `R`. Within it, you may manage the installation of `R` packages, and interact with the traditional `R` shell with the graphics popping up in a window on the right (Figure 9.1).

`R Studio` is also an editor for `R markdown`, or `Rmd` files. There is an excellent tutorial online for generated `R markdown`. Behind the scenes, a set of amazing tools called knitr and pandoc take your `markdown` code and turn it into a more standard output format. You can even select Microsoft Word (Figure 9.2).

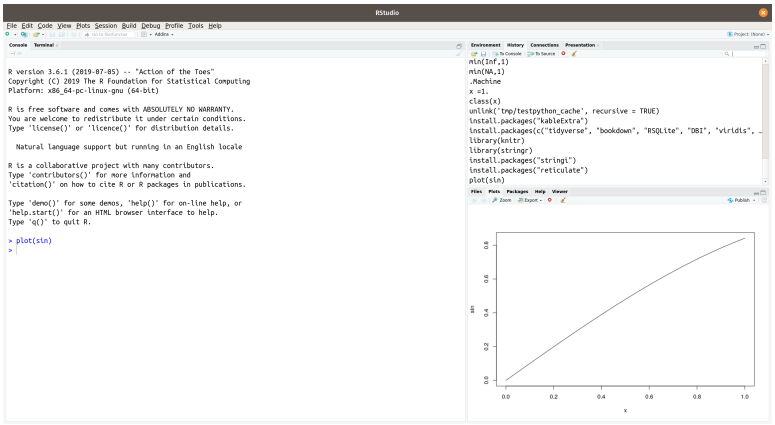


Figure 9.1: R Studio on the author's laptop running the Pop! OS 19.10 Linux distribution

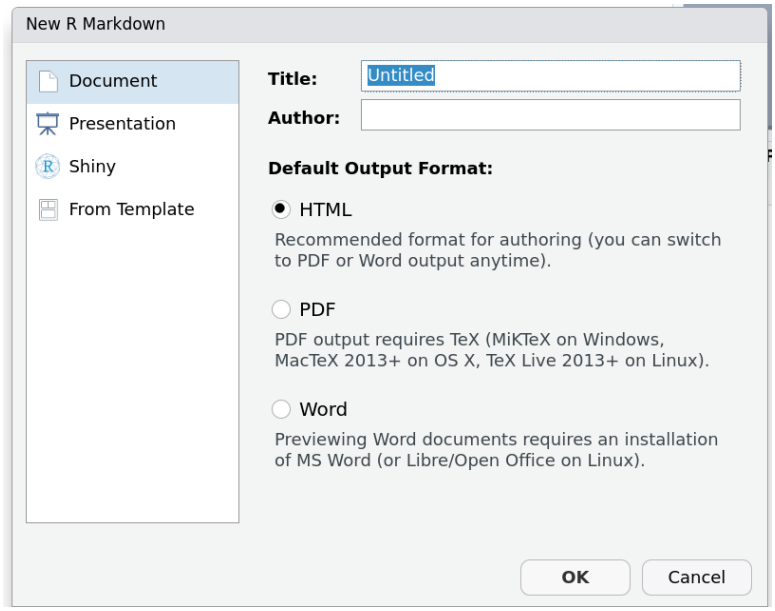


Figure 9.2: The dialog box for a new R Markdown document

 

`R Studio` also has `git` integration as long as you work on a "Project" (`File -> New Project`). You'll want to look at the `R Studio` documentation online for setting up the `git` integration.

Within an `Rmd` file, you may write `R` and `Python` code, with the latter requiring reticulate. This entire book is generated this way (Appendix E). Feel free to read the `Rmd` files behind it to see how various things get done.

My opinion is that `Rmd` files are the best solution right now for notebooks:

- They are plain text, and thus `git`-friendly.
- Support both `R` and Python in the *same* document is a big win.
- The files may be generated in any editor (because they are plain text).

## 9.2 Jupyter and Jupyter Lab

Jupyter notebooks are the most obvious notebook solution for Python (although see my comments above). `Jupyter` is much broader than just Python, however, and supports a range of languages. For example, you may write a notebook in `R` using IRKernel.

The notebooks are run through a web browser, which starts a local server. In general, it is tedious to run a notebook remotely. For example, you want to work on a notebook on the UCI cluster from your laptop. Starting it there via X11 forwarding will be a horrible experience. For effective remote work, `ssh` tunneling is needed. See here for the original link. Briefly, on the remote server, start a notebook server on an open port:

```
remote_user@remote_host$ ipython notebook --no-browser --port=8889
```

Then, from your laptop or wherever:

```
local_user@local_host$ ssh -N -f -L localhost:8888:localhost:8889 remote_user@remote_host
```

Thanks to Andy Kern (U. Oregon) for this tip.

Jupyter Lab represents the next generation interface for this technology.

I like `Jupyter` notebooks, even though I've said they are not the ideal solution. They are easy to use, generate nice output, and I just enjoy the experience (aside from having to run them via a browser). However, there is one big issue with them...

### 9.2.1 git

These notebooks contain big chunks of binary data. For us humans, it is unreadable gibberish. For `git`, it makes tracking changes over time hard. Your repository will quickly balloon in size, too. There are tools to help see the changes, but the repository size remains an issue. Other solutions, such as exporting from notebook to `markdown` and just committing the latter, are also not ideal. These issues are why I personally prefer `Rmd` files, although I still happily work in `Jupyter`–it just isn't work that generally ends up on GitHub.

 

# Chapter 10

# Generating graphics to report your research

Graphical representations of concepts and results are a major tool for communicating your science. This chapter is about making plots from data, which means that it is mostly about plotting your results. In other words, some `pdf` file that you will include in a document for submission or include in a slide. We are not talking about conceptual diagrams or schematics to illustrate concepts.

Given the topic of this chapter, many aspects are prone to subjective views. What looks good to one person does not to another. These differences of opinion are fine up to a point. No matter what aesthetic decisions are made, a good plot must have several fundamental features:

- It must faithfully represent the underlying data.
- It must be properly labelled.
- It must be detailed appropriately for the target audience. See 10.1 below.
- A good faith effort should be made to use color palettes that are friendlier to those with visual impairments.

Wilke (2019) is a good overview of making good plots from data. His book uses R, and specifically `ggplot2` plus libraries adding additional features, to make all of the plots. The book is also available online.

## 10.1 Plotting for different audiences

As a scientist, you will likely make plots for different audiences:

- Yourself, during the process of exploratory data analysis
- For a lab meeting/meeting with your advisor
- For a report for collaborators/colleagues at other institutions
- For a talk to the public
- For a presentation at a scientific conference
- For a poster
- For publication

The level of technical detail, density of visual elements, font sizes, etc., may differ for each of these target audiences. You should expect to remake the same plot multiple times when presenting to different audiences. When preparing talks and papers, expect to revise figures several times. The constant tweaking required is a good argument to generate your plots with *code* rather than with plotting tools like Excel. I also advise against the use of software like Adobe Illustrator and Inkscape for manipulating plots based on data. These

programs have a mouse-based interface and it is tedious to repeat such operations over and over again while writing a paper. (Both tools are fine for conceptual diagrams, though, and is what they are intended for.)

The process of generating figures is iterative, and often requires a lot of experimentation. If your plotting code is part of a `git` repo, the `git stash` and `git checkout` commands are helpful to undo any changes since your last commit.

## 10.2 Color schemes for plots

Be thoughtful when choosing your colors. The defaults for many of the tools discussed here are not great. There are several online sources for guidance about palettes that are friendly to color-blind viewers. The reality is that there is no perfect color scheme in the sense that you cannot guarantee that everyone will be able to distinguish every color. Rather, the point is to be conscious of this issue and not alternate red and green throughout your paper (see Eisen et al. (1998) for an example of an influential paper where red-vs-green is the key plotting technique).

When plotting continuous data using any method like a "heat map", the `viridis` palette is a good choice. It ticks a lot of boxes–perceptually uniform across its range and is at least okay for most of the common forms of color blindness, which puts it in a leading position in that regard. The palette is the default for continuous plots in `matplotlib` as of version 2.0, and it is available as an R package.

In the examples below, I show how to use `viridis` to plot discrete things in order to make what we want to focus on stand out more from the rest.

## 10.3 A tour of plotting using R and Python

You'll learn several things below, mostly by studying the code closely:

- Several data wrangling tips for both R and Python
- You'll see that the main difference between `tidyverse` and `base` R is in the defaults and not necessarily the amount of code.
- Plotting in Python isn't as bad as you've probably been told.

Again, there is no `R` vs Python here. Use the right tool for the job. The main point is to generate figures that belong in publications and that accurately reflect your data and results.

### 10.3.1 An example data set

We need a data set for plotting. Instead of using one of the built-in data sets in `R`, we will take a reduced data set from a recent paper of mine. The full version of the data set leads to Figure 10.1. The figure shows a time series resulting from a simulation of a population adapting to an environmental change. None of the details really matter for us, though. What does matter for us is that the data consists of a large number of values of a *statistic* called `tajd` in the file.

```
gz = gzfile("data/tajimasd.txt.gz",'r')
data = read.table(gz, header=T)
head(data, 20)
```

```
##    repid locus window generation         tajd      mu opt
## 1      0     0      0      40000  0.402670738 0.00025 0.1
## 2      0     0      1      40000 -0.445231328 0.00025 0.1
## 3      0     0      2      40000  0.297618074 0.00025 0.1
## 4      0     0      3      40000 -0.006280053 0.00025 0.1
## 5      0     0      4      40000 -0.530728893 0.00025 0.1
```

 

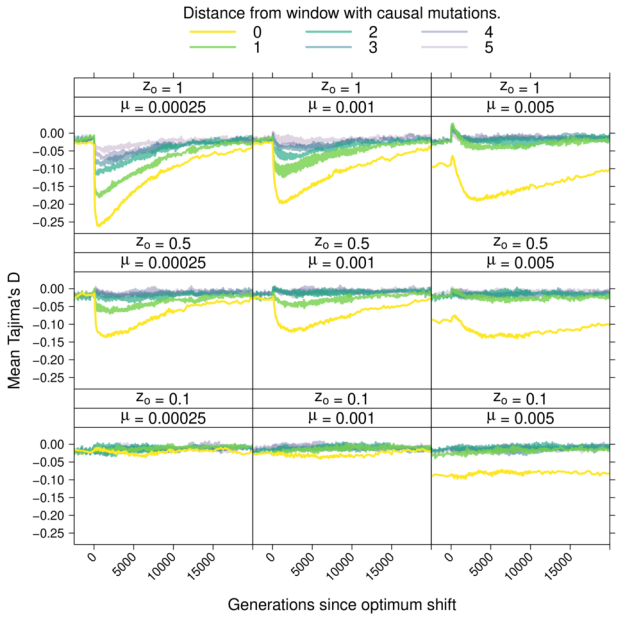


Figure 10.1: Figure 7a from Thornton (2019). The figure is reproduced here under the terms of the Creative Commons License.

```
## 6      0     0      5      40000 -0.210066368 0.00025 0.1
## 7      0     0      6      40000 -0.134733006 0.00025 0.1
## 8      0     0      7      40000 -0.010835843 0.00025 0.1
## 9      0     0      8      40000  0.166152642 0.00025 0.1
## 10     0     0      9      40000 -0.150202734 0.00025 0.1
## 11     0     0     10      40000  0.061466673 0.00025 0.1
## 12     0     1      0      40000 -0.332789222 0.00025 0.1
## 13     0     1      1      40000 -0.166023904 0.00025 0.1
## 14     0     1      2      40000  0.078974063 0.00025 0.1
## 15     0     1      3      40000 -0.031765887 0.00025 0.1
## 16     0     1      4      40000  0.313199985 0.00025 0.1
## 17     0     1      5      40000  0.288421830 0.00025 0.1
## 18     0     1      6      40000 -0.540614003 0.00025 0.1
## 19     0     1      7      40000  0.116568100 0.00025 0.1
## 20     0     1      8      40000 -0.269336968 0.00025 0.1
```

The observations of `tajd` are associated with several `conditioning variables`, or `factors`, that may or many not be relevant to our analysis:

- `repid` identifies a specific replicate simulation
- `locus` identifies a genetic locus
- `window` identifies a sub-window within a locus
- `generation` identifies a point in time
- $\mu$ and $z_o$ represent parameters of the model.

The specific analysis that we want to plot is the `mean` value of `tajd` for all combinations of $(\mu, z_o)$ for each generation. Further, for each time point, we want the mean separately for each `window` after transforming the `window` label into a *distance* from window 5. That window is the one where interesting evolution happens, and hence is the "focal" window. Given that window 5 is our focus, the plot has to make sure that the

 

data for distance zero "pops" more than for the other windows, which we accomplish with color and by also making sure that the data for `dist = 0` appears as the *top* line for each plot. We will also adjust the time units so that the time of the environmental change is the zero point on the `x` axis, which we do by subtracting a constant that you'll see below. (There is no way to just know what that constant is without reading the paper, which you don't need to do, because it is just another data wrangling issue as far as you are concerned.) Finally, we need to adjust the `x` axis limits, as there are more time points present than we actually need. None of these aesthetic details are going to happen by accident. Rather, we have to coerce the plotting tools to give us the desired result.

Figure 10.1 is based one 256 simulated data sets. The resulting file is far too big to store in the GitHub repository for this book. The file that I have provided is reduced to just the first two replicates. Because of the smaller amount of data, our plots will be noisier than Figure 10.1, but the plot details are what we are really going for here.

A word of caution before we get started. Do not expect to understand all the details of the code shown below right away. Those of you familiar with these tools may be prone to just glancing over the code and thinking that you understand it all. The reality is that there are lots of subtle details shown below. The code for the final version of the plot took many iterations, and probably many hours, to get right. Figures generally take a lot of effort to generate, and preserving them as code is very useful. Later on, when you wonder how to do something, you may remember having done it for a previous paper. You can then dig the code up from GitHub and see how to do it, saving yourself a lot of time.

You will notice that the dimensions of the graphics below differ from plot to plot. In some cases, these differences are due to different defaults for different tools. In other cases, I may have manipulated the output size. None of that really matters, as the final output size is something easily changed in the code.

## 10.3.2 R

### 10.3.2.1 base

Now, we'll go through the exercise of generating a version of Figure 10.1 using `base` R graphics. Murrell (2005) is a standard reference for `base` graphics. The citation is to the first edition, and it has been updated multiple times since I got my copy.

First, we need to turn the `window` column into a `distance` value, representing the distance of a window from window 5:

```
data$dist <- abs(data$window - 5)
```

Now, we obtain the mean of the statistic for every combination of factors:

```
agg <- aggregate(x=data[c("tajd")], by=data[c("mu","opt","generation","dist")], FUN=mean)
```

Finally, we can shift time such that generation 50,000 is zero:

```
agg$scaled_time <- agg$generation - 10*5e3
```

The result of the above data wrangling is a `data.frame` that we can use for plotting:

```
head(agg)
```

```
##        mu opt generation dist        tajd scaled_time
## 1 0.00025 0.1      40000    0  0.06509943      -10000
## 2 0.00100 0.1      40000    0 -0.13380347      -10000
## 3 0.00500 0.1      40000    0 -0.03057952      -10000
## 4 0.00025 0.5      40000    0 -0.10106159      -10000
## 5 0.00100 0.5      40000    0  0.04539441      -10000
```

 

```
## 6 0.00500 0.5      40000     0 -0.03040589      -10000
```

We are going to manually set our x-axis limits, so we'll keep them as a global variable:

```
XLIM=c(-2500,4*5e3)
```

To generate the colors, we will take a number of colors from the `viridis` color scheme equal to the number of lines in each plot. They will come out darkest to lightest and we will modify the list so that darker colors are more transparent (smaller values for `alpha`).

```
library(viridis)
```

```
## Loading required package: viridisLite
```

```
ICOLORS=viridis(length(unique(as.factor(data$dist))))
COLORS=array()
for(i in 1:length(ICOLORS))
{
    ncolor = rgb(as.integer(col2rgb(ICOLORS[i])[1,])/255,
                 as.integer(col2rgb(ICOLORS[i])[2,])/255,
                 as.integer(col2rgb(ICOLORS[i])[3,])/255,
                 alpha=i/length(ICOLORS))
    COLORS[i]=ncolor
}
```

Figure 10.2 shows the output of the following code, generated using the `coplot` function. The plot is fine for an exploratory analysis, but it is rather far from publication-ready. The way that the conditioning variables are displayed takes up far too much space. We also don't have a legend showing what our lines mean. The use of the `panel` function to draw the lines themselves is a bit tedious and we'd prefer that sort of grouping to be more automatic.

```
coplot(tajd ~ scaled_time | as.factor(mu)*as.factor(opt), data=agg,
       xlim=XLIM,
       subscripts = TRUE,
       panel=function(x, y, subscripts, ...)
       {
           sset <- agg[subscripts,]
           i <- 1
           # Plot lines is reverse order of variable
           # value such that value 0 is plotted
           # last/on top.
           for (d in rev(sort(unique(sset$dist))))
           {
               ssetd = subset(sset, dist == d)
               lines(ssetd$scaled_time, ssetd$tajd, col=COLORS[i])
               i <- i+1
           }
       },
     xlab=c("Generations since optimum shift",expression(mu)),
     ylab=c("Mean Tajima's D", expression(z[o])))
```

It is simpler to make these types of plots with packages where line grouping within the conditioning variables is automatic, such as `ggplot2` and `lattice`. Before showing how to use them, we will redo our analysis using the `tidyverse` package `dplyr`.

 

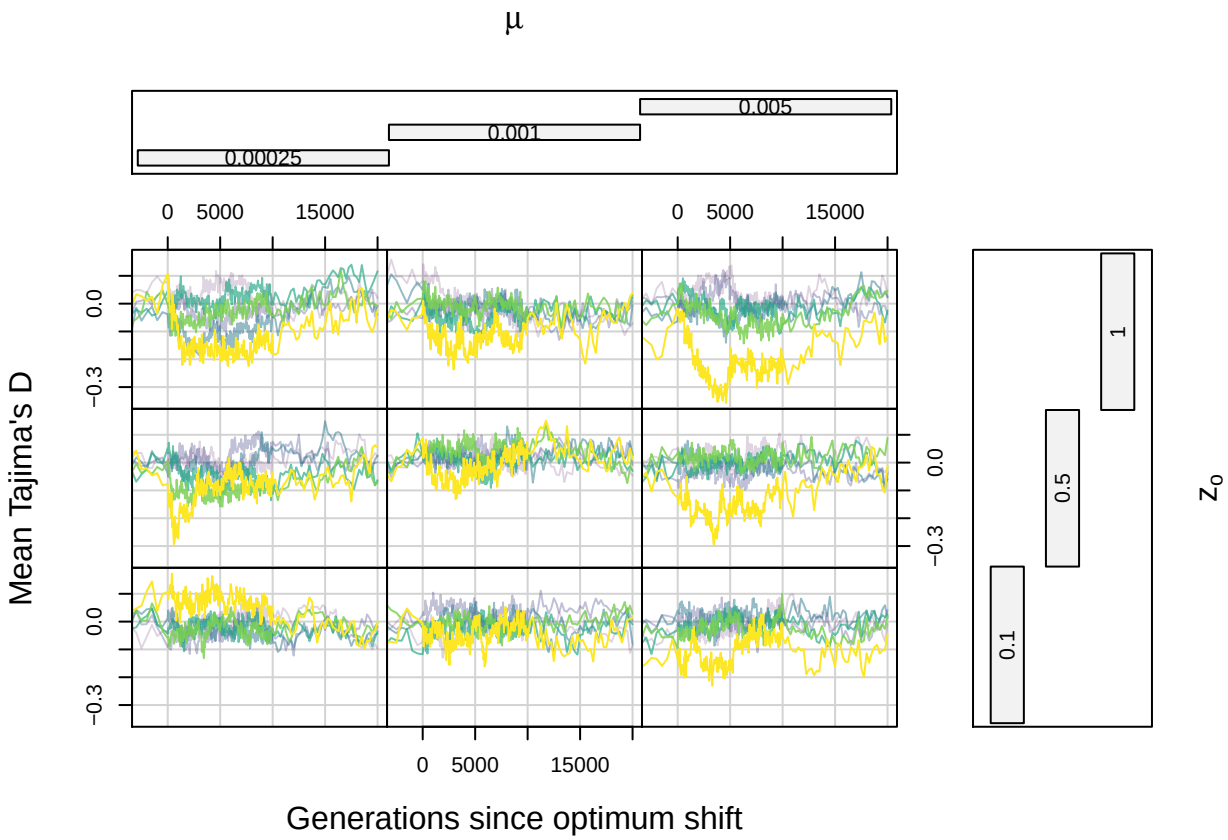


Figure 10.2: The output from base R's coplot

 

#### 10.3.2.2 Processing the data using the tidyverse

The following code results in a data frame with contents very similar to our `agg` object above. One nice feature of `dplyr` is that it is easy to name the outputs of our analysis. For example, we name the output of our summary function `meand`. Above, the output of `mean(tajd)` was still named `tajd` in `agg` when using the `aggregate` function.

We also take the time here to make sure that we have properly ordered `factor` columns in our output. These columns are used to generate the plotting order of the conditioning variables.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(readr)

gz = gzfile("data/tajimasd.txt.gz",'rb')
data = read_delim(gz, "\t") %>%
       mutate(dist = factor(abs(window-5),
                            levels=rev(sort(unique(abs(window-5)))))) %>%
       group_by(generation, dist, mu, opt) %>%
       summarise(meand = mean(tajd)) %>%
       mutate(scaled_time = generation - 10*5e3) %>%
       mutate(fopt = factor(opt, levels=c(1,0.5,0.1)))
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   repid = col_double(),
##   locus = col_double(),
##   window = col_double(),
##   generation = col_double(),
##   tajd = col_double(),
##   mu = col_double(),
##   opt = col_double()
## )
```

```
## `summarise()` regrouping output by 'generation', 'dist', 'mu' (override with `.groups` argument)
```

```
head(data)
```

```
## # A tibble: 6 x 7
## # Groups:   generation, dist, mu [2]
##   generation dist       mu   opt    meand scaled_time fopt
##        <dbl> <fct>   <dbl> <dbl>    <dbl>       <dbl> <fct>
## 1      40000 5     0.00025   0.1 -0.0508       -10000 0.1
## 2      40000 5     0.00025   0.5  0.0386       -10000 0.5
## 3      40000 5     0.00025   1    0.0124       -10000 1
```

 

```
## 4      40000 5      0.001      0.1   0.0329       -10000 0.1
## 5      40000 5      0.001      0.5  -0.0914       -10000 0.5
## 6      40000 5      0.001      1    -0.00820      -10000 1
```

#### 10.3.2.3 ggplot2

Wilke (2019) and Wickham and Grolemund (2017) are good references on plotting using `ggplot2`.

`ggplot2` is great for quick visualizations of your data:

```
library(ggplot2)
p <- ggplot(data=data,aes(x=scaled_time, y=meand, group=dist)) +
    facet_grid(fopt~mu) +
    geom_line(aes(color=dist,lty=dist)) +
    xlim(XLIM[1], XLIM[2]) +
    theme_bw() +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    xlab("Generations since optimum shift") +
    ylab("Mean Tajima's D")
p
```

```
## Warning: Removed 90 row(s) containing missing values (geom_path).
```


Figure 10.3: Exploratory data analysis with ggplot2

This code, shown in Figure 10.3, is a good enough exploratory data analysis for most lab meetings. The axes

 

are readable and labelled while the plotting factors are semi-labelled. The colors are not great, so I attempted to use line styles (`lty`) to make it a bit more readable. The result isn't great, but at least there is a legend.

Do not let the relative brevity of the above code block hide the fact that some specific manipulations were necessary to get the desired output:

- The order in which lines are plotted in each panel depends on their sort order as `factors`.
- The ordering of the rows and columns also depends on their sort order as `factors`.

Let's work on getting the figure closer to publication-ready, which will also show us how to improve the labelling.

The following code generates the figure using `ggplot` and the results are shown in 10.4.

```
p <- ggplot(data=data,aes(x=scaled_time, y=meand, group=dist)) +
    facet_grid(fopt~mu,
               labeller=label_bquote(rows = z[0] == .(fopt),
                                     cols = mu == .(mu))) +
    geom_line(aes(color=dist)) +
    xlim(XLIM[1], XLIM[2]) +
    scale_color_manual(values=COLORS) +
    theme_bw() +
    theme(legend.position='top') +
    theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
    theme(panel.grid.major = element_blank(),
          panel.grid.minor = element_blank()) +
    labs(color="Distance from window with causal mutations") +
    xlab("Generations since optimum shift") +
    ylab("Mean Tajima's D")
p
```

```
## Warning: Removed 90 row(s) containing missing values (geom_path).
```

#### 10.3.2.4 lattice

The `lattice` package is the subject of the book by Sarkar (2008). It is based on the `grid` package extending the `base` functionality (Murrell (2005)). This is the package that I used to generate Figure 10.1 and the code from the paper is shown below. It is not much more complex than the `ggplot` versions. The major subtlety is that you need to be very careful with custom keys (legends). It is possible to get the color order wrong, the label order wrong, or both.

```
library(lattice)

KEY=list(space="top",columns=3,
         title="Distance from window with causal mutations.",
         cex.title=1,
         lines=list(lwd=rep(3,length(COLORS)),col=rev(COLORS)),
         just=0.5,
         text=list(as.character(rev(sort(unique(data$dist))))))

STRIP=strip.custom(strip.names = TRUE,sep=" = ",
                   var.name = c(expression(mu),expression(z[o])),bg=c("white"))

tajdPlot = xyplot(meand ~ scaled_time| as.factor(mu)*as.factor(opt),
                  group=factor(dist,levels=sort(unique(data$dist))),
```

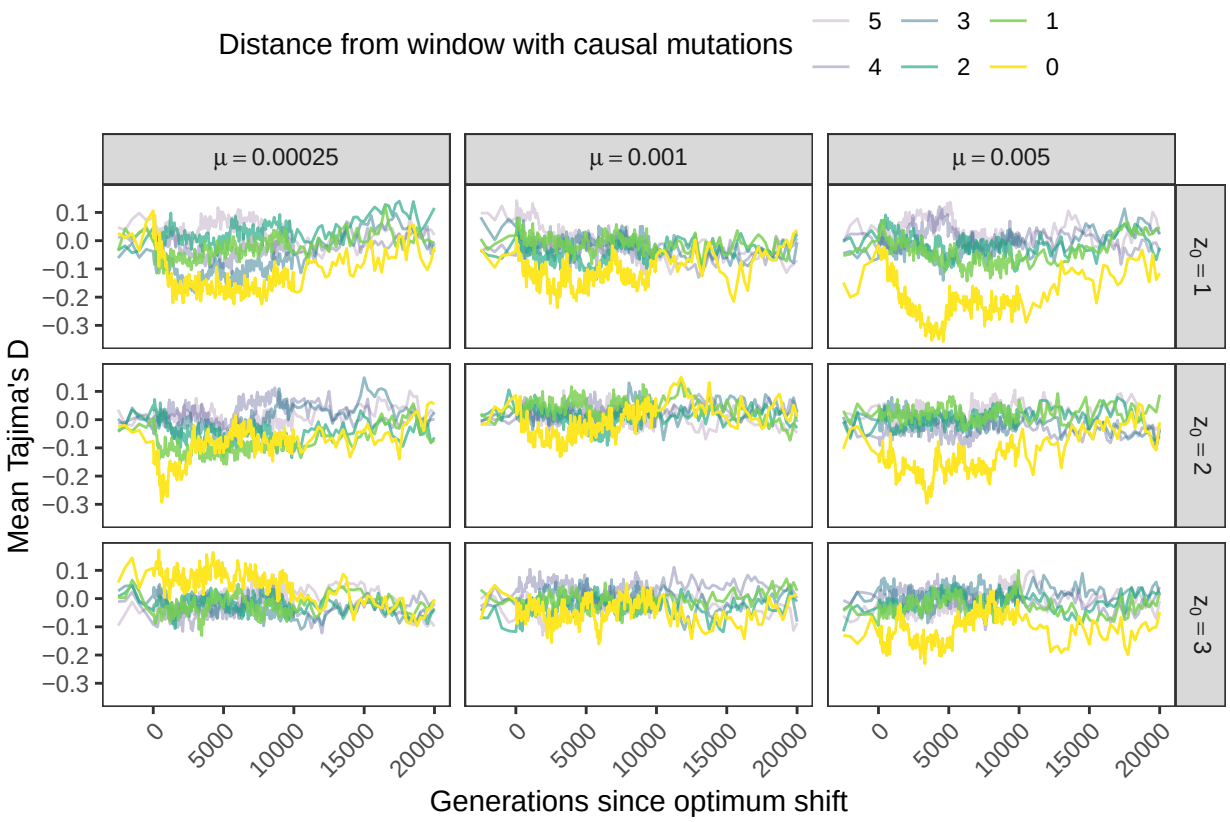


Figure 10.4: The figure generated with ggplot2

 

```
                type='l',data=data,
                par.settings=simpleTheme(col=COLORS),
                key=KEY, lwd=3,
                xlab="Generations since optimum shift",
                ylab="Mean Tajima's D",
                xlim=XLIM,
                scales=list(cex=0.75,alternating=F,x=list(rot=45)),
                strip=STRIP)
```

The result of this code is shown in Figure 10.5.

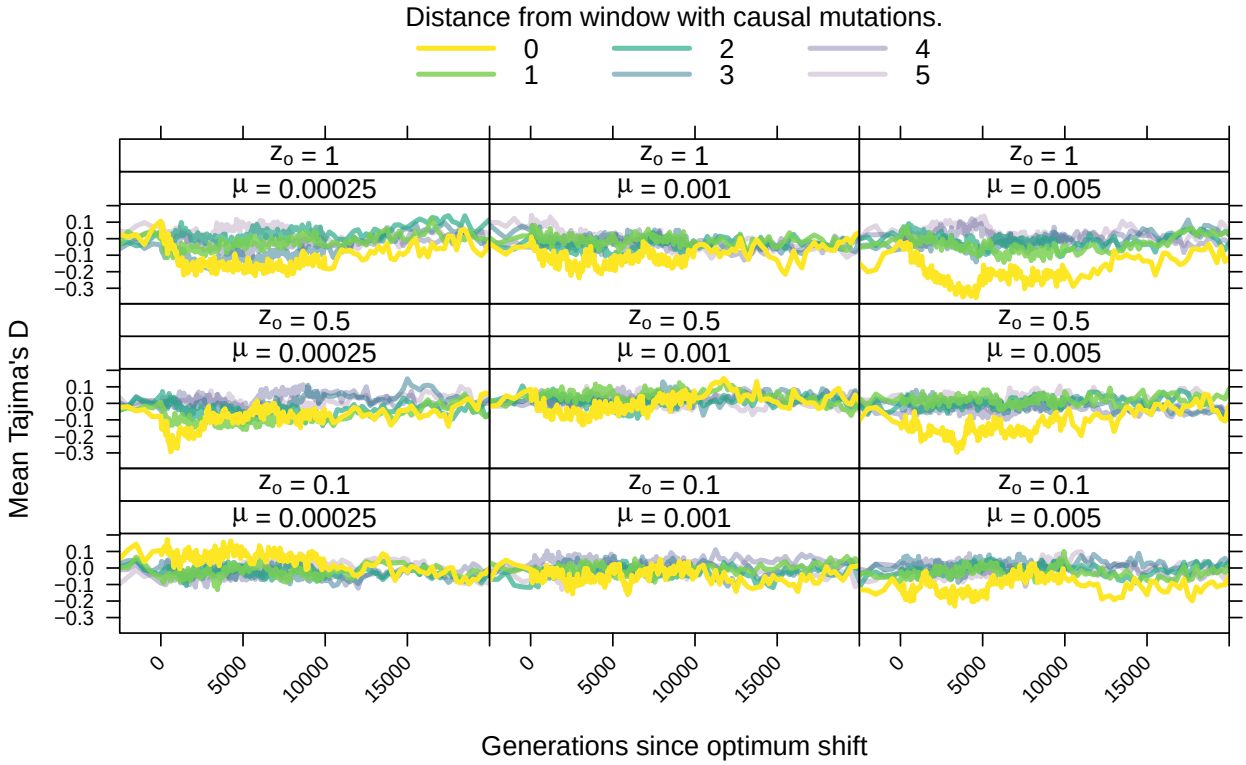


Figure 10.5: The figure generated with the lattice package, dplyr, and viridis

### 10.3.3 Python

While R is the more popular choice for generating plots, Python has a powerful set of libraries for plotting. Given that the final result satisfies the criteria of a good plot, the language that you choose is not especially important. The only catch is that, if you work in Python and plot in R, then you need a means of getting data between the two languages.

#### 10.3.3.1 Manipulating our data using pandas.

Before we start, we need to read in our data and summarise is as we did in R. The following code generates a data frame object that we can use for plotting. The code uses `pandas` (see Chapter 6.15) and isn't any more complex than the `dplyr` code above.

 

```
import pandas as pd
import numpy as np

# Note: the calls to drop are optional.

input_data = pd.read_csv('data/tajimasd.txt.gz', sep='\t', compression='gzip')
input_data.drop(labels=['locus','repid'], axis=1, inplace=True)
input_data['scaled_time'] = input_data['generation'] - 10*5e3
input_data['dist'] = np.abs(input_data['window'] - 5)
data = input_data.groupby(['scaled_time','mu','opt','dist']).mean().reset_index()
data.drop(labels=['generation'], axis=1, inplace=True)
```

#### 10.3.3.2 matplotlib

All plotting in Python is based on matplotlib, which provides an object-oriented toolkit for static graphics. Like `base` R, `matplotlib` is a rather low-level solution. In some respects, it is simpler than `base` R, but it is more complex in other cases. I personally like how multi-panel layouts are done. Each panel is an instance of the `matplotlib` `Axes` class, meaning that it is a variable that can be manipulated.

A nice `matplotlib` feature is the ability to use raw LaTeX code in string literals. It is also possible to use LaTeX to render the fonts in a plot, but I do not do so here.

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
```

First, we define our axis limits for the plot like we did in R.

```
XLIM=(-2500,4*5e3)
YLIM=(-0.4,0.20)
```

Now, we have to construct our plot layout. The example uses `GridSpec` to manage the panels. The following code sets up a 3-by-3 list of lists of `Axes` for the plotting. The three outer lists correspond to the rows of our plot and the elements within each row are the columns. We need every panel to have the same `x` and `y` axis ranges, and we will specify that when creating the panels. We will define the axis ranges for the upper left panel (row 0, column 0) and tell the remaining panels that they share its axis limits.

```
fig = plt.figure(figsize=(6,6)) # Units are inches

# Generate a 3x4 plot layout
# The 4th column gives us room to make a legend
gs = GridSpec(3, 4, width_ratios=(1,1,1,0.5))

# Create our first panel
row0 = [fig.add_subplot(gs[0,0])]

# Set up the parameters to indicate
# axis sharing
shareargs = {'sharex':row0[0], 'sharey':row0[0]}

# Create the remaining axis objects
row0.extend([fig.add_subplot(gs[0,1], **shareargs),
             fig.add_subplot(gs[0,2], **shareargs)])

row1 = [fig.add_subplot(gs[1,0], **shareargs),
```

 

```
        fig.add_subplot(gs[1,1], **shareargs),
        fig.add_subplot(gs[1,2], **shareargs)]

row2 = [fig.add_subplot(gs[2,0], **shareargs),
        fig.add_subplot(gs[2,1], **shareargs),
        fig.add_subplot(gs[2,2], **shareargs)]

# Hide all the axis ticks and labels
# for all panels except those along
# left and bottom edges
for i in row0 + row1:
    i.get_xaxis().set_visible(False)

for i in row0[1:] + row1[1:] + row2[1:]:
    i.get_yaxis().set_visible(False)

# The various x = foo(...) below grab the return
# values of the calls so that they aren't
# printed to the page

# Manually specify where the axis tick marks are,
# the axis font size, and rotation for the x axis.
# This code is over-riding the defaults b/c I like
# the result more.
for i in row2:
    x = i.set_xticks([0, 5000, 1e4, 1.5e4])
    x = i.set_xticklabels([0, 5000, 10000, 15000], rotation=30,
                          fontsize='small')

for i in [row0[0], row1[0], row2[0]]:
    x = i.set_yticks([-0.3, -0.15, 0.0, 0.15])
    x = i.set_yticklabels([-0.3, -0.15, 0.0, 0.15],
                          fontsize='small')
```

Having generated the axis objects, we will group them into a list of lists and then set the axis limits. The list of lists makes the actual plotting steps more convenient.

```
axes = [row0, row1, row2]
x = axes[0][0].set_xlim(XLIM)
x = axes[0][0].set_ylim(YLIM)
```

When we generate our plot, it will help to be able to map our plotting parameters back to row and column indexes. It is straightforward to use a `dict` to build these mappings:

```
optindexes = {j:i for i,j in enumerate(reversed(sorted(data.opt.unique())))}
mutindexes = {j:i for i,j in enumerate(sorted(data.mu.unique()))}
```

It is now time to set up our colors. We will pull them from the `viridis` color palette. In `matplotlib`, the color maps are objects with values that may be retrieved using floats in the interval `[0, 1]`. We want good color separation, so we will pull colors uniformly (on a linear scale) from close to the entire range:

```
ncolors = len(data.dist.unique())
COLORS = [plt.cm.viridis(f) for f in np.linspace(0.01,0.99,ncolors)]
```

 

The colors are simply tuples of RGB and alpha values:

```
COLORS[:2]
```

```
## [(0.269944, 0.014625, 0.341379, 1.0), (0.252194, 0.269783, 0.531579, 1.0)]
```

The colors go from darkest to lightest and we want alpha transparency value to increase along the list (*i.e.* brighter colors are more opaque). Let's make a new color list where with a gradient of decreasing transparency. We will also reverse the list of colors so that it corresponds more naturally to our `dist` variable:

```
COLORS = [(*i[:3], j) for i,j in zip(COLORS, np.linspace(0.2, 1, ncolors))]
COLORS = COLORS[::-1]
```

Using the fourth column in our grid, we use the second row to set up a legend:

```
legend_axis = fig.add_subplot(gs[1,3], sharey=axes[0][0])
legend_axis.set_title("Window\ndistance", fontsize='small')
x = legend_axis.axis('off')
```

Having set things up, we can now generate the plot itself, which proceeds via simple grouping operations using `pandas`. In the code, the string literals contain LaTeX code to generate the symbols, which is one of my favorite `matplotlib` features. The front-to-back placement of lines is controlled by the `zorder` parameter.

The code follows and the output is shown in Figure 10.6.

```
import matplotlib.lines as mpl_lines

for n, g in data.groupby(['mu', 'opt']):
    c = mutindexes[n[0]]
    r = optindexes[n[1]]
    for d, dg in g.groupby(['dist']):
        axes[r][c].set_title(r'$\mu = $ {}'.format(n[0]) +', ' +
                             r'$z_o = $ {}'.format(n[1]),
                             fontsize='small')
        axes[r][c].plot(dg.scaled_time, dg.tajd,
                        # Annoyance: group name is one
                        # variable, so it isn't in a tuple
                        color=COLORS[d],
                        # Lower zorder = line further to the back
                        zorder=len(COLORS)-d)
axes[1][0].set_ylabel("Mean Tajima's D")
axes[2][1].set_xlabel("Generations since optimum shift")

# Now, fill in our legend
lpos = np.linspace(YLIM[0]*0.95,YLIM[1]*0.95, len(COLORS))
for c,y in zip(COLORS, lpos):
    legend_axis.add_line(mpl_lines.Line2D((0.1,0.5),
                                          (y+0.0075,y+0.0075),
                                          color=c))

x = legend_axis.set_xlim(0, 1)
for i,y in enumerate(lpos):
    legend_axis.text(0.6, y, i, fontsize='small')

plt.show()
```

 

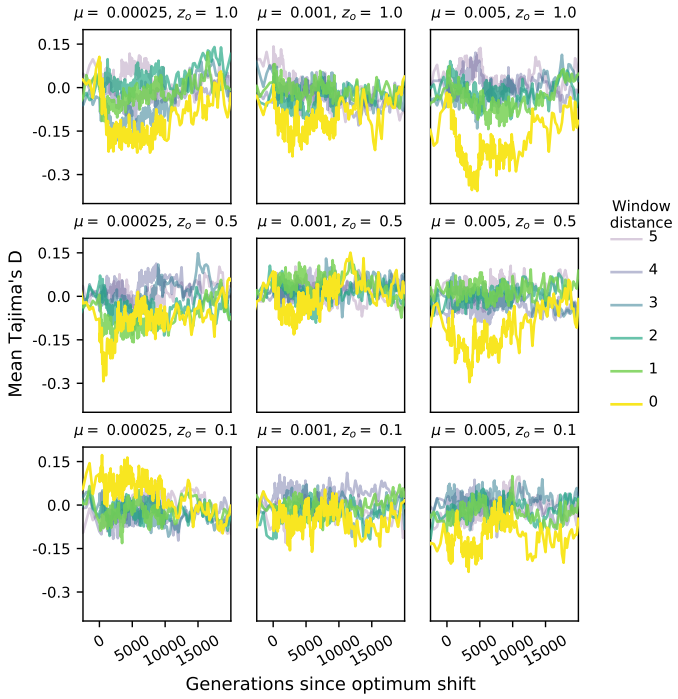


Figure 10.6: Figure done in Python using matplotlib and pandas

The major problem with Figure 10.6 is the manual legend. If we need to revise our figures and put the legend at another location, all of the `GridSpec` bits need to be changed.

#### 10.3.3.3 seaborn

The seaborn package provides higher-level abstractions for plotting in Python. The following code generates Figure 10.7.

```
import seaborn as sns
hue_dict = {i:COLORS[i] for i in data.dist.unique()}
g = sns.FacetGrid(data, row='opt', col='mu', xlim=XLIM,
                  hue='dist', palette=hue_dict,
                  row_order=reversed(sorted(data.opt.unique())),
                  hue_order=len(COLORS) - data.dist.unique() - 1)
g = g.map(sns.lineplot, "scaled_time", "tajd")
x = g.set_axis_labels("Generations since optimum shift", "Mean Tajima's D")
x = g.set_titles(r'$\mu = $' + " {col_name}, " + r'$z_o = $' + " {row_name}")
x = g.add_legend(title="Window distance")
plt.show()
```

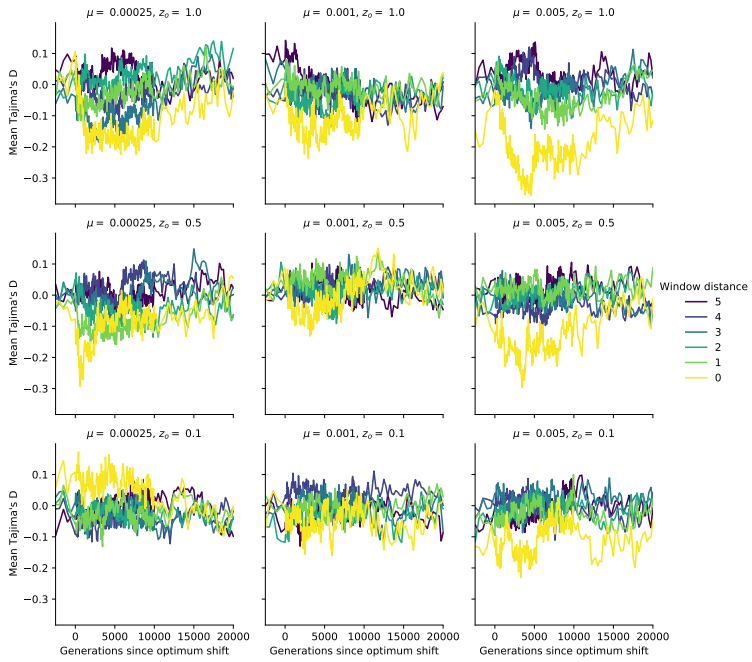


Figure 10.7: Figure made with the Python seaborn library.

The code is very compact, but still required some tricks:

- We are using a custom color setup, so we had to map our `dist` factor to a color via `hue_dict`. Note that the final plot doesn't respect the alpha transparency that we set up earlier. I am not sure why, but would consider that an issue.
- The row/column orders are based on standard sorting, so we had to specify the orders. The `row_order` was easy. We didn't need to mess with `col_order` because the default sorting gives the desired result.
- The `hue_order` is non-obvious at first, but it specifies the order in which the lines get added to a panel. Our line factor, `dist` starts at 0, but we want that value to be plotted last, so that the line is on top.

 

The ordering is just like how `zorder` works (see above). Therefore a simple bit of logic specifies the orders:

```
data.dist.unique()
```

```
## array([0, 1, 2, 3, 4, 5])
```

```
len(COLORS) - data.dist.unique() - 1
```

```
## array([5, 4, 3, 2, 1, 0])
```

The results from `seaborn` are impressive. The plot is probably publication-quality, or at least very close, and the amount of code needed is small. The fonts are probably a bit small. The legend placement may not be ideal, but it is hard-coded in to a `center right` position for this type of plot. I have read the `seaborn` code and see why they made this decision, but it is possible that a future release will allow legend placement "anywhere" like a standard `matplotlib` plot. The issue is that the layout of these multi-panel plots is a bit fiddly, which you got a hint of in the previous section.

#### 10.3.3.4 ggplot for Python

There have been a few attempts at a `ggplot` for Python. The one that seems to be sticking is plotnine. I have not used it, but am pointing it out in case it helps someone.

## 10.4 Generating graphics for publication

For a publication, you must save your figures as files. You need to consider the file format for output. Most people I know are in the habit of saving all figures as `pdf` files. However, journals typically want high-resolution `TIFF` files.

I will guess that many of the images shown in this chapter will have trouble being saved to file as-is. Font sizes may not work out when you save to a certain size, etc.. In R, it is common that what you see in a graphics window and what gets saved to a file differ subtly. I have experienced this many times on systems using the X11 graphics system (Linux and R on older versions of `macOS`). In other words, working on your plot by repeatedly saying `source('plot.R')` and looking at what pops out on the screen may not reflect what you get when you finally save it to a `pdf`.

What follows is a list of loose guidelines:

- Make your graphics files big, with big fonts, and size them down in your document. In R, I tend to make 10 by 10 inch plots with 18 point fonts as a starting point. I then size them down in LaTeX via the `\scale` parameter passed to `\includegraphics`.
- Prefer *lossless* image formats!! From R, this includes `pdf`, `png`, and `tiff`. For the last two, set the output resolution to 300 dpi. For Python, `pdf` and `png` work. LaTeX users may also consider `ps` and `eps` (Postscript and Encapsulated Postscript, respectively).
- Prefer formats that your target journal accepts. This will rule out `png` for sure, and quite often `pdf`. The two Postscript formats are accepted by several journals.
- Prefer file formats that are small enough to include in your documents. This often rules out 300 dpi `tiff`. For web pages, `png` is ideal, and `pdf` generally not allowed.

### 10.4.1 Image Magick

You are now certainly confused, as we have ruled out every format! That's right, there is no objectively perfect file format, bringing us to our final guideline:

- Learn how to reliably convert between formats without losing quality.

 

ImageMagick exists to help with this last guideline. It is a command-line Swiss Army knife for image manipulation. Yes, you could use various graphical tools to convert. Apple's Preview on `macOS` is great for this, but it cannot be easily automated.

For example, the following `bash` loop will convert all of your `pdf` files into 300 dpi `tiff` while also flattening all layers and replacing any transparent backgrounds with white:

```
for i in *.pdf
do
    n=`basename $i .pdf`
    convert -density 300 -flatten -background white -compress lzw $i $n.tiff
done
```

Oh, we also compressed the file using the `lzw` algorithm, which is accepted by most journals that I work with. Loops like this one are hard to beat when you have to convert possibly dozens of graphics files.

`ImageMagick` can also help you with other issues, like if your files are rejected for having the wrong color space encoding. In the past, graphics output from R were rejected from PLoS journal's automatic "QC" system for this reason. To make figures that passed, we set the R scripts to output 600 dpi `tiff` and then run this loop:

```
#!/bin/bash

# Credit to Jaleal Sanjak for working this out for one or more
# of his PhD papers.

for i in $(ls *.tif | grep -v _compressed)
do
    n=`basename $i .tif`
    convert $i -set colorspace RGB -layers flatten -alpha off -compress lzw -depth 8 \
    -density 600 -adaptive-resize 4500x2400  $n"_compressed".tif

    convert $n"_compressed".tif $n"_compressed".pdf
done
```

In hindsight, the 600dpi output may have been overkill, but it worked. The final conversion to `pdf` was for inclusion in our LaTeX document.

On some systems, you may get errors when trying to `convert` from `pdf` (or `ps`/`eps`) to other formats. This error is in place because of a security flaw in `ghostscript`. See here for how to enable conversion (link is to Stack Overflow).

## 10.5 Generating graphics for talks

All of the above applies, but you can get away with smaller files like `png`.

Expect to have to modify your figures for talks. You are all but guaranteeing a bad talk if you take figures directly from your paper and drag them into Powerpoint. Papers from publications are often far too detailed, and the fonts too small, to be visible from the back of the room. Every talk is a job interview, so don't be lazy here. For the figures shown in this chapter, I would reduce them to a single row or perhaps even to a single panel. The idea that you may need to change your figures **yet again** reinforces the importance of generating them with code that is under version control.

 

## 10.6 Hints for Google Docs

If you write your documents using this tool, then you need to make sure your image files are all on the Google Drive using the same account as your Docs. The reason is the Docs only allows you to drag small image files from your local machine into a document. A larger file will get rejected, but then will work fine if you select "Insert image from Drive". Even if you manage to drag in a local file, Docs may automatically downsize it, resulting in a loss of image quality.

Google Docs seems to prefer `png` above all else. I expect that these guidelines apply to the other Google tools, too.

## 10.7 Interactive graphics

Academia has long been subjugated by the process of publication via dead trees. Even with most/all of our journals being accessed *exclusively* online, that access is still most often based around the display of a `pdf` file with static graphics. It may sometimes be the case that a graphic can be made more effective by making it interactive. Several tools exist to do this, and they generally result in an HTML output with some embedded JavaScript. You may want to check out the following tools:

- shiny for R.
- plotly for R.
- bokeh for Python.
- plotly for Python.
- holoviews for Python.

All of these tools are designed to run in the appropriate notebook system for each language (see Chapter 9). They also allow export to `html` files that you may display online, including through a GitHub site. Here is an example that I generated using `holoviews` and a Jupyter notebook.

 

# Chapter 11

# Databases using sqlite3

Before we begin, I want to say that this chapter contains material whose utility you may not realize right now, but may want to revisit later. The methods discussed here solve a lot of problems, but you may not "be there" yet with your own research. Also, do not give into the temptation to use the approaches described here instead of domain-specific approaches for your specific problems. For example, if bedtools can process your data using one command, then definitely do that! Likewise, if `R` or Python libraries already do exactly what you need, then use them. This chapter is ultimately about what to do when you need to build a custom wheel.

This chapter describes how to store and manipulate data stored in *databases* rather than in *data frames*. These databases may be on disk or in memory, but most often the former. What we have been calling a data frame is referred to as a table in database jargon. Data bases are useful in many cases:

- When your data set is too large to hold in memory.
- When your data set consists of a very large number of tables and your analysis needs to integrate all of them.
- When you want to process your data using tools written in several different languages. The database can act as a common file format.
- When you have to reformat data into a tabular format for a custom analysis. (This is a common need. It may be worth considering having the final file be a database rather than a "flat" text file.)

Data bases are binary file formats optimized for fast retrieval. We will talk about the sqlite data base engine, or `sqlite3` after its current version. `sqlite3` is notable because you may "administer" databases (*e.g.* create and delete them) without needing privileged access on your machine. Thus, you can create them on the cluster. The trade-off is that `sqlite3` is not as powerful (fast) as some of its open-source competitors MySQL or PostgreSQL.

Databases have their own language called the Structured Query Language, or SQL. The various database programs all use slightly different variants of SQL. The main point of this chapter is that you can use the `tidyverse` library `dplyr` in place of learning SQL syntax and writing the queries manually.

## 11.1 Examples using R

The `R` code in this chapter has some additional dependencies that you will want to install:

- `DBI`
- `RSQLite`
- `dbplyr`

I will use explicit namespace references in the example code shown here. What function comes from which package will be a bit too murky otherwise.

Let's take our data set from Chapter 10 and output it as an `sqlite3` database:

```
x = read.table(gzfile("data/tajimasd.txt.gz"), header=T)
conn = DBI::dbConnect(RSQLite::SQLite(), "tajimasd.sqlite3")
RSQLite::dbWriteTable(conn, "data", x)
RSQLite::dbDisconnect(conn)
```

That was rather easy. One hint is to use `sqlite3` as the suffix instead of `db`. By default, Google Drive doesn't like the latter extension.

The next bit is the real magic. We will use our database to do the same aggregation as in Chapter 10.

```
conn = dplyr::src_sqlite("tajimasd.sqlite3")
```

```
## Warning: `src_sqlite()` is deprecated as of dplyr 1.0.0.
## Please use `tbl()` directly with a database connection
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

```
dbtable = dplyr::tbl(conn, 'data')
query = dbtable %>%
       mutate(dist = abs(window-5)) %>%
       group_by(generation, dist, mu, opt) %>%
       summarise(meand = mean(tajd, na.rm=T)) %>%
       mutate(scaled_time = generation - 10*5e3)

data = dplyr::collect(query)

data$dist <- factor(data$dist, levels=rev(sort(unique(data$dist))))
data$opt <- factor(data$opt, levels=c(1, 0.5, 0.1))
```

The `query` variable looks a lot like what we did in the graphics chapter. The only difference is that we don't deal with the `factor` business there. That line is the main thing to take away from this chapter:

**You can write SQL queries using dplyr, and it will generate the correct SQL syntax for you when you `collect` the query results. The query is executed on the database side, and therefore not in memory!!!**

That last sentence is really important: when your analysis is too big to do in RAM, then you may be able to do it in a database using the *exact same* toolkit.

There is a caveat, of course. The SQL that `dplyr` writes requires that there is a direct mapping of a `dplyr` verb into SQL syntax. If no such mapping is possible, you'll get strange errors. One such example is `factor`, which is an `R` concept that has no analog in SQL. Thus, we assign the `factor` levels after getting the results back from the query.

## 11.2 Examples using Python

In Python, dumping a data frame to a database is straightforward using `pandas`:

```
import pandas as pd
import numpy as np
import sqlite3
```

 

```
x = pd.read_csv('data/tajimasd.txt.gz', sep='\t', compression='gzip')
conn = sqlite3.connect("tajimasd.sqlite3")
x.to_sql('data', conn)
conn.close()
```

To retrieve and analyze data using `pandas`, we need to use plain SQL queries:

```
conn = sqlite3.connect("tajimasd.sqlite3")
query = 'select avg(tajd) as meand, ' + \
        'abs(window - 5) as dist, ' + \
        'generation - 50000 as scaled_time ' + \
        'from data ' + \
        'group by generation, dist, mu, opt'
agg = pd.read_sql(query, conn)
conn.close()
```

## 11.3 Recommendations

In general, I recommend `dplyr` for the *processing* of databases and either `R` or Python for making them. For Python, SQL Alchemy provides higher-level abstractions for database interaction, meaning that you don't need to write the raw SQL. However, I personally find it very complex and greatly prefer `dplyr` because I don't have to learn anything new! (Other than how to connect to the database and the table, which are trivial operations.)

## 11.4 Joining tables

This chapter doesn't cover the "power" feature of databases, which the *join*. In words, a join may take the form of "take all rows from table `A` where the values in column `C` equal the values in column `C` from table `B`".

You may perform joins on in-memory data frames in both R, via `dplyr`, and in Python, using `pandas`. From an `sqlite3` database, you can write the same kind of joins using `dplyr` and they will be executed out of memory.

Joins are a big topic. There are many flavors of join. It is well worth learning the `dplyr` syntax for in- and out-of- memory use. The `pandas` syntax is powerful, too.

 

# Chapter 12

# Dependency management

This chapter covers the tools for *dependency management*, meaning the installation of software with all of the other software needed (the *dependencies*). This chapter is in an odd location of the book. We should set you up with `conda` early in the class, but describing it first seems to break the flow of earlier chapters.

Let's refer to a piece of software as a package. `conda` and `brew` exist as third-party package managers, or tools to install packages, meaning that they exist alongside the native package manager for your operating system (if one exists).

Tools like `conda` or `homebrew` are attempting to solve a very difficult problem. In general, it is quite difficult to support thousands of different packages for multiple operating systems. In the case of R and Python packages, it is also very difficult to provide reliable packages for libraries for those languages that are compatible with multiple versions of each language. Thus, none of these solutions are really perfect and things will go wrong from time to time. However, I expect that you will find them useful and they are certainly much easier than maintaining installations of everything you need from source.

This chapter tries to solve a handful of interrelated problems:

- Most of our students/postdocs use Mac OS, which has no package manager for Unix tools.
- You may need to install software on Unix machines where you don't have "super user" privileges.
- You may need to have multiple versions of the same software on your system, or some other complexity requiring you to "wall off" the installation of one set of tools from another.

## 12.1 conda

Reasons to use `conda`:

- You do not need privileged ("superuser") access to your machine. By default, things are installed in your `$HOME` directory.
- On systems like Apple's `macOS`, which has no package manager, you will now have one.
- Packages are compiled in a consistent manner helping to prevent incompatibilities.
- You may have multiple versions of the same software on your system via *environments*.
- It is available for Linux, `macOS`, and Windows

For those familiar with Python's `virtualenv` feature, `conda` is a generalization of that concept to include software outside of the Python ecosystem.

The major alternative to `conda` is homebrew, which is available for Linux and `macOS`.

### 12.1.1 What it does and does not do

Briefly, `conda` allows you to install packages, have multiple versions of those packages installed in different environments, and "dump" a list of these environment contents to file. These files can be used to recreate the same environment on another machine of the *same* operating system. `conda` does **not** allow the precise replication of environments between *different* operating systems. For example, you cannot create an environment on `macOS` and then expect to replicate it exactly on the Linux environment of the UCI HPC. Simply put, there is no guarantee that the same versions of the same packages are available at the same time for all operating systems (and with those packages having the same versions of the same dependencies).

### 12.1.2 Getting conda

By `conda`, we really mean `miniconda`. Get the latest Python 3 installer here and follow the instructions. Accept the defaults, unless you have good reason not to.

If you use the Ubuntu subsystem on Windows 10, then you actually want to download the *Linux* installer!

### 12.1.3 Channels

There are different organizations, called *channels*, that supply `conda` packages. An important channel is Bioconda. Follow the instructions at their site to set up your channels correctly–the order matters!

If you use packages from `Bioconda`, please cite Grüning et al. (2018).

### 12.1.4 Using conda

The main documentation site is quite good. It is useful to search for the *command reference* page within the main docs.

Key commands to know are:

- `conda install` to install packages
- `conda list` to list packages in the current environment
- `conda remove` to remove packages from the current environment
- `conda update` to update packages in the current environment

#### 12.1.4.1 Environments

Managing environments is described in detail in the documentation, which would be good to look over right now, so that you have an idea of what is possible.

In this section, I want to discuss examples of how environments may be useful. In order for your results to be reproducible, you need to be doing all of your analyses with the same versions of the various packages that you are using. Therefore, one common use of environments is to create different environments for different projects.

You can also use environments to test if your results are sensitive to changing the versions of packages. Briefly, you may `clone` your current environment into a new one, then run `conda update` to update the new environment. Once that is done, rerun some or all of your analyses and see if the results are the same or not. If not, maybe you can figure out why not. (This procedure could actually reveal bugs in your code!) Based on what you see, you can decide what to do next. You can decide to stick with the old or the new environment, and delete the other.

If your project is sufficiently complex, it may be useful to dump your environment's contents to a file and save that file as part of the `git` repository for the project. While it is unlikely that you will be able to recreate this precise environment in the future, you'll have an idea of where to start.

 

A final use case involves your institution's compute cluster. If you are allowed to install `conda` there, it may be a useful way to manage dependencies for yourself. There are several advantages to being able to do this. Here at UCI, it is possible to install `conda` for yourself as a `module`, which we discuss in the lab portion of the course.

## 12.2 brew

**TBD**

## 12.3 Containers (Docker, etc.)

**TBD**

 

# Appendix A

# Necessary hardware and operating system

This class requires a laptop for the labs. The idea is that using your own machine simulates a more realistic experience of getting research computing done.

Life will be easier if you have a machine that either is, or can emulate, a Unix-like environment. In order of ease of use, the following operating systems will work:

- Linux. Basically any up-to-date release of any distribution. Please be sure to have Python 3 installed. Most distros default to Python 2, and 3 must be installed separately.
- Apple's `macOS`. Preferably High Sierra or later. A serious complication is that the file system is not case-sensitive, meaning that `File.txt` and `file.txt` are treated as the same thing. You absolutely *must* have the appropriate version of `Xcode` installed for your system and have taken the extra steps to install the command line tools for that version of `Xcode`. It is up to you to look up the required steps for that. Section A.2 details issues that you may run across with this platform.
- Windows. If you use Windows, you will have the toughest time. I think, but am not totally sure, that you will need Windows 10 (or later). There are a variety of "terminal emulators" for Windows, and it will be up to you to find one that works. Windows 10 supports the "Windows Subsystem for Linux", which provides things like a `bash` shell in an environment based on the Ubuntu Linux distribution.

## A.1 Issues with Linux

This is the home team OS for scientific computing. Most of your problems will be interacting with collaborators using Microsoft Office, Adobe products, etc..

## A.2 Issues with `macOS`

This OS is slowly being locked down so that applications installed from sources other than the App Store may run into issues.

### A.2.1 Lack of clarity about the best way to install certain tools

We will recommend `conda` in this course for package management, but that isn't always the right answer. For example, `conda` contains packages for `Rstudio`, LaTeX, and some others, but I **do not** recommend installing

them this way onto `macOS`. Rather, prefer the `.dmg` installers for these tools or, in some cases, you need to use `brew`.

### A.2.2 Third party software after major updates

When you make a major update from one "named" release to the next (*i.e.*, `Mojave` to `Catalina`), you should expect third party software (stuff not installed via the "App Store") to break. It won't happen all the time, and what does break could change each time, but stuff does break! An incomplete list of things to monitor includes:

- MacTex, which is the typical way to get LaTeX onto an `macOS` machine.
- Your `conda` and `brew` installations.
- Inkscape
- Any software installed into `/usr/local` may be deleted!! `brew`, for example, and most/all Unix software installed from source that didn't go into a `conda` environment, assuming that you installed `conda` into your user's home directory.

### A.2.3 Xcode

You need to remember to update `Xcode` with each major `macOS` update, too! And, don't forget the command line tools.

### A.2.4 Python 3

As of `Catalina` (10.15), `Xcode` ships with Python 2. Future versions (10.16+) will do something different with all Unix "scripting" languages like Python and Perl, but it remains to be seen what that is. It seems likely that they will not be installed by default, meaning you'll need to use something like `conda` or `brew` to install them.

### A.2.5 Unreliable behavior of programs using X11/XQuartz

Unix programs with a graphical interface rely on the X11 graphics toolkit. On `macOS`, this is provided by XQuartz. It is not always straightforward to get programs using X11 to work on current `macOS` releases. For example, if you install `imagemagick` via `conda`, then the `display` program (which displays graphics on the screen) probably won't work. Instead, you will have to use the `open` command on the file, which will open `Preview`. Other `imagemagick` binaries that don't require `X11`, such as `convert`, work just fine in my experience. (I have the same issues installing these programs via `brew`.)

## A.3 Issues with Microsoft Windows

I don't have enough experience with this platform to have any content here, but there are certainly issues!

 

# Appendix B

# Ergonomics

Make sure you have a proper chair. Get your monitor set up to the right height. You probably really like the freedom of a laptop, but don't forget that it is really bad for you in the long run! The monitor is making you crane your neck, and the keyboard has you twisting your wrists at a funny angle. The taller you are, and the broader your shoulders, the worse all of this is. All of these issues will catch up to you, eventually.

## B.1 Keyboard

The keyboard is your primary input mechanism for writing and coding. As such, it is a primary ergonomics consideration. A good keyboard should:

- Prevent twisting at the wrists
- Have all meta buttons (`Alt`, `Cntrl`, etc.) available on both sides
- Have good key "travel" meaning that your fingers don't bottom out while typing

Laptop keyboards usually don't meet all of these criteria. For example, Apple keyboards only have `Cntrl` buttons on the left. All commercially-available keyboards also have a staggered key layout, meaning that the keys "fan out" to the left or right from the center. There is no real reason for this design. I am a fan of the alternative "ortholinear" design where the keys are in straight columns. However, all ortholinear keyboards are boutique products, and most are sold as DIY kits requiring assembly and programming, which leads me to one of the few product recommendations in this book. For my desktop computer, I use the Ergodox EZ keyboard and I use a Planck EZ whenever possible with my laptop. These ortholinear keyboards are shown in Figure B.1. I recognize that the price is steep, but I highly recommend these keyboards, especially the `Ergodox`. Both take some getting used to, but they have made a huge difference in managing my issues with carpal tunnel syndrome. Both keyboards are completely programmable, and you can control the mouse cursor with them, too.

For desktop computers, I like the offerings from Das Keyboard for traditional keyboard layouts. The `Das` is a nice mechanical keyboard which feels good and is not too loud.

If you do get a mechanical keyboard like the `Das`, `Ergodox` or `Planck`, please be considerate of your lab mates! Try to avoid loud key switches. For the record, I use Cherry Brown in my `Ergodox` and Kailh Gold in my `Planck`. Both feel great and are pretty quiet, but not totally silent.

No matter what keyboard you use, there is one ergonomic improvement that everyone can make, which is to remap the `Caps Lock` key to something useful. That key is completely useless for pretty much anyone. If you use the `vim/neovim` editors, then remap it to `Esc` if you haven't done so already. Your wrists will thank you.

Figure B.1: The Erdogox EZ (left) and the Planck EZ (right)

 

# Appendix C

# Other programming languages of interest

## C.1 C

C is a compiled language. It is high-performance, has a small standard library, and is the *de facto* standard of the Unix world. R and Python are both implemented in C.

At first glance, C looks like it has a shallow learning curve, but that is just a reflection of the small standard library. The truth is that you need to manually manage all resources and do so in away that doesn't "leak" resources when errors occur.

## C.2 C++

C++ is a compiled language that initially added objects to C. Now, it seems to support nearly every programming idiom on the planet. It is a high-performance language with a very steep learning curve. It is my language of choice for performance-oriented code. Unlike C, resource management may be handled a bit more safely via objects. Otherwise, it is a much more complex language, although also a much richer one. Scott Meyer's books are essential texts for anyone working in C++.

## C.3 julia

julia is a kind of a hybrid of interpreted and compiled languages. If your work involves solving systems of ODEs or PDEs, then you should investigate julia. Colleagues of mine who do that sort of thing tell me that it makes Matlab irrelevant. As an added bonus, Chris Rackauckas developed the ODE/PDE parts of julia during his PhD work here at UCI.

## C.4 rust

rust is a high-performance compiled language originally developed by Mozilla. It intends to compete with C and C++ for high-performance programming. The language is new, but quite intriguing. Like C++, the learning curve will be steep, but there is a lot to like in what I've read. Expect to see biological software written in rust.

# Appendix D

# Advanced git

There is nothing here yet. The stuff that should be here has the potential to cause a `gitastrophe`, meaning that you may do damage to you repo that is hard/impossible to undo. More in a future version...

# Appendix E

# Technical details behind this book

This book is generated using the bookdown flavor of R markdown, or `Rmd`. While `Rmd` files are usually generated using R Studio, I instead use the following work flow:

- The .Rmd files are edited using neovim with the following plugins related to `markdown`
  - vim-pandoc/vim-pandoc
  - vim-pandoc/vim-pandoc-syntax
  - vim-pandoc/vim-rmarkdown

  These plugins are from vim-pandoc.
- `git` integration in `neovim` is provided by tpope/vim-fugitive.
- The book is built from .Rmd files using GNU make.
- Schematic diagrams are generated using tikz and auto-processed to `png` via a combo of custom Makefile rules to execute `pdflatex` and Image Magick.
- The bibliographic entries are Bibtex files generated by outputting a folder from Paper Pile, which is the only non-free software used.

This book is generated on a Linux operating system, most likely Pop! OS version 19.10 or later.

You can get the source code for this book here.

# Bibliography


Vince Buffalo. *Bioinformatics Data Skills: Reproducible and Robust Research with Open Source Tools*. O'Reilly Media, Inc., July 2015. ISBN 9781449367510.

M B Eisen, P T Spellman, P O Brown, and D Botstein. Cluster analysis and display of genome-wide expression patterns. *Proc. Natl. Acad. Sci. U. S. A.*, 95(25):14863–14868, December 1998. ISSN 0027-8424. doi: 10.1073/pnas.95.25.14863.

Björn Grüning, Ryan Dale, Andreas Sjödin, Brad A Chapman, Jillian Rowe, Christopher H Tomkins-Tinch, Renan Valieris, Johannes Köster, and Bioconda Team. Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nat. Methods*, 15(7):475–476, July 2018. ISSN 1548-7091, 1548-7105. doi: 10.1038/s41592-018-0046-7.

Jerome Kelleher, Alison M Etheridge, and Gilean McVean. Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Comput. Biol.*, 12(5):e1004842, May 2016. ISSN 1553-734X, 1553-7358. doi: 10.1371/journal.pcbi.1004842.

Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. March 2013.

Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009. ISSN 1367-4803, 1367-4811. doi: 10.1093/bioinformatics/btp324.

Wes McKinney. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 2 edition edition, October 2017. ISBN 9781491957660.

Paul Murrell. *R Graphics (Chapman & Hall/CRC The R Series)*. Chapman and Hall/CRC, 1 edition edition, July 2005.

Luciano Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media, 1 edition edition, August 2015. ISBN 9781491946008.

Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R (Use R!)*. Springer, 1st edition, August 2008. ISBN 9780387759685.

Adam Siepel. Challenges in funding and developing genomic software: roots and remedies. *Genome Biol.*, 20 (1):147, July 2019. ISSN 1465-6906. doi: 10.1186/s13059-019-1763-7.

Kevin R Thornton. Polygenic adaptation to an environmental shift: temporal dynamics of variation under gaussian stabilizing selection and additive effects on a single trait. *Genetics*, Early online, 2019. ISSN 0016-6731. doi: 10.1101/505750.

Jake VanderPlas. *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, 1 edition edition, December 2016. ISBN 9781491912058.

Hadley Wickham. The Split-Apply-Combine strategy for data analysis. *Journal of Statistical Software*, 40(1), 2011.

Hadley Wickham and Garrett Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, 1 edition edition, January 2017. ISBN 9781491910399.

Claus O Wilke. *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O'Reilly Media, 1 edition edition, April 2019. ISBN 9781492031086.

 