Statistical Computing using R and Python

Susan Vanderplas

January 13, 2023

Table of contents

Preface

Acknowledgements

The cover of this book is an amalgam of different images by the lovely @allison_horst, which are released under the cc-by 4.0 license. I have modified them to remove most of the R package references and arrange them to represent the topics covered in this book.

Laptop icon used in the tab/logo created by Good Ware - Flaticon

Throughout this book, I have borrowed liberally from other online tutorials, published books, and blog posts. I have tried to ensure that I link to the source material throughout the book and provide appropriate credit to anyone whose examples I have used, modified, or repurposed. Special thanks to the tutorials provided by Posit/RStudio and the tidyverse project.

How to Use This Book

Content Overload!

This book is designed to demonstrate introductory statistical programming concepts and techniques. It is intended as a substitute for hours and hours of video lectures - watching someone code and talk about code is not usually the best way to learn how to code. It's far better to learn how to code by … coding.

I hope that you will work through this book week by week over the semester. I have included comics, snark, gifs, YouTube videos, extra resources, and more: my goal is to make this a collection of the best information I can find on statistical programming.

In most cases, this book includes **way more information** than you need. Everyone comes into this class with a different level of computing experience, so I've attempted to make this book comprehensive. Unfortunately, that means some people will be bored and some will be overwhelmed. Use this book in the way that works best for you - skip over the stuff you know already, ignore the stuff that seems too complex until you understand the basics. Come back to the scary stuff later and see if it makes more sense to you.

Book Format Guide

I've made an effort to use some specific formatting and enable certain features that make this book a useful tool for this class.

Buttons/Links

The book contains a number of features which should help you navigate, use, improve, and respond to the textbook.

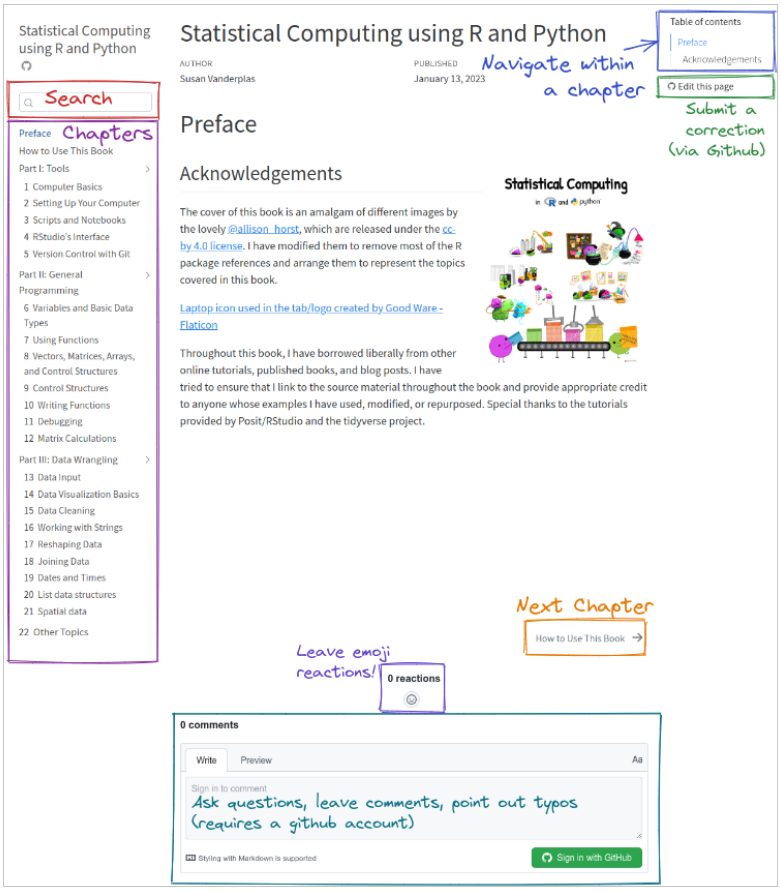


Figure 1: Textbook features, menus, and interactive options

Special Sections

! Warnings

These sections contain things you may want to look out for: common errors, mistakes, and unfortunate situations that may arise when programming.

Demonstrations

These sections demonstrate how the code being discussed is used (in a simple way).

Examples

These sections contain illustrations of the concepts discussed in the chapter. Don't skip them, even though they may be long!

Try it out

These sections contain activities you should do to reinforce the things you've just read. You will be much more successful if you read the material, review the example, and then try to write your own code. Most of the time, these sections will have a specific format:

0.0.0.0.1 * Problem
The problem will be in the first tab for you to start with

0.0.0.0.2 * R Solution
A solution will be provided in R, potentially with an explanation.

0.0.0.0.3 * Python Solution
A solution will be provided in Python as well.
In some cases, the problem will be more open-ended and may not adhere to this format, but most try it out sections in this book will have solutions provided. I **highly**

recommend that you attempt to solve the problem yourself before you look at the solutions - this is the best way to learn. Passively reading code does not result in information retention.

! Essential Reading

These sections may direct you to additional reading material that is essential for understanding the topic. For instance, I will sometimes link to other online textbooks rather than try to rehash the content myself when someone else has done it better.

Learn More

These sections will direct you to additional resources that may be helpful to consult as you learn about a topic. You do not have to use these sections unless you are 1) bored, or 2) hopelessly lost. They're provided to help but are not expected reading (Unlike the essential reading sections in red).

i Notes

These generic sections contain information I may want to call attention to, but that isn't necessarily urgent or a common error trap.

Advanced

These sections are intended to apply to more advanced courses. If you are taking an introductory course, feel free to skip that content for now.

Expandable Sections

These are expandable sections, with additional information when you click on the line

This additional information may be information that is helpful

but not essential, or it may be that an example just takes a LOT of space and I want to make sure you can skim the book without having to scroll through a ton of output.

> **Another type of expandable note**
>
> Answers or punchlines may be hidden in this type of expandable section as well.

Part I

Part I: Tools

This part of the textbook provides an overview of the different tools we will be using: R, python, quarto, markdown, pandoc, consoles, and so on. It can be a bit confusing at first, especially if you're not familiar with how your computer works, where files are stored, and different ways to tell your computer what to do.

Chapter 1 gives you some important background material about how a computer functions.

Chapter 2 tells you exactly what software you need to install for the rest of this textbook.

Chapter 3 discusses the different ways we can talk to R and python, and the pros and cons of each.

1 Computer Basics

1.1 Objectives

- Know the meaning of computer hardware and operating system terms such as hard drive, memory, CPU, OS/operating system, file system, directory, and system paths
- Understand the basics of how the above concepts relate to each other and contribute to how a computer works
- Understand the file system mental model for computers

1.2 Hardware

Here is a short 3-minute video on the basic hardware that makes up your computer. It is focused on desktops, but the same components (with the exception of the optical drive) are commonly found in cell phones, smart watches, and laptops.

When programming, it is usually helpful to understand the distinction between RAM and disk storage (hard drives). We also need to know at least a little bit about processors (so that we know when we've asked our processor to do too much). Most of the other details aren't necessary (for now).

- Chapter 1 of Python for Everybody - Computer hardware architecture

1.3 Operating Systems

Operating systems, such as Windows, MacOS, or Linux, are a sophisticated program that allows CPUs to keep track of multiple programs and tasks and execute them at the same time.

1.4 File Systems

For this class, it will probably be important to distinguish between local file storage (C:/ drive , /user/your-name/ , or /home/your-name/) and network/virtual file systems, such as OneDrive and iCloud. Over time, it has become harder to ensure that you are working on a local machine, but working "in the cloud" can cause odd errors when programming and in particular when working with version control systems[1].

You want to save your files in this class to your **physical hard drive**. This will save you a lot of troubleshooting time.

Evidently, there has been a bit of generational shift as computers have evolved: the "file system" metaphor itself is outdated because no one uses physical files anymore. This article [1] is an interesting discussion of the problem: it makes the argument that with modern search capabilities, most people use their computers as a laundry hamper instead of as a nice, organized filing cabinet.

Regardless of how you tend to organize your personal files, it is probably helpful to understand the basics of what is meant by a computer **file system** – a way to organize data stored on a hard drive. Since data is always stored as 0's and 1's, it's important to have some way to figure out what type of data is stored in a specific location, and how to interpret it.

That's not enough, though - we also need to know how computers remember the location of what is stored where. Specifically, we need to understand **file paths**.

When you write a program, you may have to reference external files - data stored in a .csv file, for instance, or a picture. Best

[1] To disable onedrive sync for certain windows folders, use this guide

practice is to create a file structure that contains everything you need to run your entire project in a single file folder (you can, and sometimes should, have sub-folders).

For now, it is enough to know how to find files using file paths, and how to refer to a file using a relative file path from your base folder. In this situation, your "base folder" is known as your **working directory** - the place your program thinks of as home.

1.5 System Paths

When you install software, it is saved in a specific location on your computer, like C:/Program Files/ on , /Applications/ on , or /usr/local/bin/ on . For the most part, you don't need to keep track of where programs are installed, because the install process (usually) automatically creates icons on your desktop or in your start menu, and you find your programs there.

Unfortunately, that isn't sufficient when you're programming, because you may need to know where a program is in order to reference that program – for instance, if you need to pop open a browser window as part of your program, you're (most likely) going to have to tell your computer where that browser executable file lives.

To simplify this process, operating systems have what's known as a "system path" or "user path" - a list of folders containing important places to look for executable and other important files. You may, at some point, have to edit your system path to add a new folder to it, making the executable files within that folder more easily available.

> **How To Modify System Paths**
>
> How to set system paths (general)
> Operating-system specific instructions cobbled together from a variety of different sources:
>
> - On Windows

- On Mac
- On Linux

If you run across an error that says something along the lines of

- could not locate xxx.exe
- The system cannot find the path specified
- Command Not Found

you might start thinking about whether your system path is set correctly for what you're trying to do.

If you want to locate where an executable is found (in this example, we'll use `git`), you can run `where git` on windows, or `which git` on OSX/Linux.

Some programs, like RStudio, have places where you can set the locations of common dependencies. If you go to Tools > Global Options > Git/SVN, you can set the path to git.

1.6 References

2 Setting Up Your Computer

2.1 Objectives

- Set up RStudio, R, Quarto, and python
- Be able to run demo code in R and python

2.2 Installation Process

In this section, I will provide you with links to set up various programs on your own machine. If you have trouble with these instructions or encounter an error, post on the class message board or contact me for help.

1. Download and run the R installer for your operating system from CRAN:

   - Windows: https://cran.rstudio.com/bin/windows/base/
   - Mac: https://cran.rstudio.com/bin/macosx/
   - Linux: https://cran.rstudio.com/bin/linux/ (pick your distribution)

   If you are on Windows, you should also install the Rtools4 package; this will ensure you get fewer warnings later when installing packages.

   More detailed instructions for Windows are available here

2. Download and install the latest version of python 3

   - Then, install Jupyter using the instructions here
   - Additional instructions for installing Python 3 from Python for Everybody if you have trouble.

3. Download and install the latest version of RStudio for your operating system. RStudio is a integrated development environment (IDE) for R - it contains a set of tools designed to make writing R code easier.
4. Download and install the latest version of Quarto for your operating system. Quarto is a command-line tool released by RStudio that allows Rstudio to work with python and other R specific tools in a unified way.

The following steps may be necessary depending on which class you're in. If you want to be safe, go ahead and complete these steps as well.

5. Install git using the instructions here. Consult the troubleshooting guide if you have issues. If that fails, then seek help in office hours.
6. Install LaTeX and rmarkdown:
    - Launch R, and type the following commands into the console:

```
install.packages(c("tinytex", "knitr", "rmarkdown", "quarto"))
library(tinytex)
install_tinytex()
```

Your turn

Open RStudio on your computer and explore a bit.

- Can you find the R console? Type in `2+2` to make sure the result is `4`.
- Run the following code in the R console:

```
install.packages(
  c("tidyverse", "rmarkdown", "knitr", "quarto")
)
```

- Can you find the text editor?
    - Create a new quarto document (File -> New File -> Quarto Document).

- Paste in the contents of this document.
- Compile the document (Ctrl/Cmd + Shift + K) and use the Viewer pane to see the result.
- If this all worked, you have RStudio, Quarto, R, and Python set up correctly on your machine.

3 Scripts and Notebooks

3.1 Objectives

- Understand the different modes that can be used to interact with a programming language
- Identify which mode and language is being used given a screenshot or other contextual information
- Select the appropriate mode (interactive, script, notebook) for a given task given considerations such as target audience, human intervention, and need to repeat the analysis.

3.2 A Short History of Talking to Computers

The fundamental goal of this chapter is to learn how to talk to R and Python. In the very beginning, people told computers what to do using punch cards [2]. This required that you have every step of your program and data planned out in advance - you'd submit your punch cards to the computer, and then come back 24-72 hours later to find out you'd gotten two cards out of order. Dropping a tray of punch cards was … problematic.

Thankfully, we're mostly free of the days where being a bit clumsy could erase a semester of hard work. As things grew more evolved and we got actual monitors and (eventually) graphical interfaces, we started using interactive terminals (**interactive mode**) to boss computers around.

Your Turn - Interactive Mode

3.2.0.1 R

Open RStudio and navigate to the Console tab. You can issue commands directly to R by typing something in at the `>` prompt.
Try typing in `2+2` and hit enter.

3.2.0.2 Python

Open RStudio and navigate to the Terminal tab. This is your computer's 'terminal' - where you tell the computer what to do.
First, we have to tell it what language we'd like to work in - by default, it's going to work in Batch (Windows), Zsh (Mac), or Bash (Linux). Luckily, we can avoid these and tell the computer we want to work in python by typing in `python3` or `python` (depending on how your computer is set up). This will launch an interactive python session (**ipython**).
You should get a prompt that looks like this: `>>>`
Type in `2+2` and hit enter.

Interactive mode is useful for quick, one-off analyses, but if you need to repeat an analysis (or remember what you did), interactive mode is just awful. Once you close the program, the commands (and results) are gone. This is particularly inconvenient when you need to run the same task multiple times. For example, each day I may want to pull the weather forecast and observed weather values from the national weather service using the same commands. I don't want to manually re-type them each day!

To somewhat address this issue, most computing languages allow you to provide a sequence of commands in a text file known as a **script**. Scripts are typically meant to run on their own - they may perform computations, format data and save it, scrape data from the web… the possibilities are endless, but they are typically meant to run without the person running the script having to read all of the commands.

Your Turn - Script Mode

1. Download scripts.zip and unzip the file.
2. Open a system terminal in the directory where you unzipped the files.

3.2.0.3 Windows

Open the folder. Type cmd into the location bar at the top of the window and hit enter. The command prompt will open in the desired location.

3.2.0.4 Mac

Open a finder window and navigate to the folder you want to use. If you don't have a path bar at the bottom of the finder window, choose View > Show Path Bar. Control-click the folder in the path bar and choose Open in Terminal.

3.2.0.5 Linux

Open the folder in your file browser. Select the path to the folder in the path bar and copy it to the clipboard. Launch a terminal and type `cd`, and then paste the copied path. Hit enter. (There may be more efficient ways to do this, but these instructions work for most window managers).

3. Now, let's try out script mode in R and Python!

3.2.0.6 R

This assumes that the R binary has been added to your system path. If these instructions don't work, please ask for help or visit office hours.
In the terminal, type `Rscript words.R dickens-oliver-twist.txt`
You should get some output that looks like this:

```
user@computer:~/scripts$ Rscript words.R dickens-oliver-twist.txt
text
```

```
 the  and        to   of    a  his   in   he  was
8854 4902 4558 3767 3763 3569 2272 2224 1931 1684
```

3.2.0.7 Python

This assumes that the python binary has been added to your system path. If these instructions don't work, please ask for help or visit office hours.
In the terminal, type `python3 words.py` and hit Enter. You will be prompted for the file name. Enter `dickens-oliver-twist.txt` and hit Enter again.
You should get some output that looks like this:

```
user@computer:~/scripts$ python3 words.py
Enter file:dickens-oliver-twist.txt
the 8854
```

Scripts, and compiled programs generated from scripts, are responsible for much of what you interact with on a computer or cell phone day-to-day. When the goal is to process a file or complete a task in exactly the same way each time, a script is the right choice for the job.

However, when working with data, we sometimes prefer to combine scripts with interactive mode - that is, we use a script file to keep track of which commands we run, but we run the script interactively. About 60% of my day-to-day computing is done using R or python scripts that are run interactively.

Your Turn - Interactive Scripts

If you haven't already, download scripts.zip and unzip the file.
Open RStudio and use RStudio to complete the following tasks.

3.2.0.8 R

1. Use RStudio to open the `words-noinput.R` file in the `scripts` folder you downloaded and unzipped.
2. What do you notice about the appearance of the

file? Is there an icon in the tab to tell you what type of file it is? Are some words in the file highlighted?

3. Copy the path to the scripts folder.
   OS Specific Instructions: Windows, Mac, Linux

4. In the Console, type in `setwd("<paste path here>")`, where you paste your file path from step 3 between the quotes. Hit enter.

5. In the `words-noinput.R` file, hit the "source" button in the top right. Do you get the same output that you got from running the file as a script from the terminal? Why do you think that is?

6. Click on the last line of the file and hit Run (or Ctrl/Cmd + Enter). Do you get the output now?

7. Click on the first line of the file and hit Run (or Ctrl/Cmd + Enter). This runs a single line of the file. Use this to run each line of the file in turn. What could you learn from doing this?

3.2.0.9 Python

1. Use RStudio or your preferred python editor to open the `words-noinput.py` file in the `scripts` folder you downloaded and unzipped.

2. What do you notice about the appearance of the file? Is there an icon in the tab to tell you what type of file it is? Are some words in the file highlighted?

3. Copy the path to the scripts folder.
   OS Specific Instructions: Windows, Mac, Linux

4. In the Console, type in `setwd("<paste path here>")`, where you paste your file path from step 3 between the quotes. Hit enter.

5. In the `words-noinput.py` file, hit the "source" button in the top right. Do you get the same output that you got from running the file as a script from the terminal? What changes?

6. Click on the first line of the file and hit Run (or Ctrl/Cmd + Enter). This runs a single line of the file. Use this to run each line of the file in turn. What do you learn from doing this?

Using scripts interactively allows us to see what is happening in the script step-by-step, and to examine the results during the program's evaluation. This can be beneficial when applying a script to a new dataset, because it allows us to change things on the fly while still keeping the same basic order of operations.

3.3 Writing Code for People

One problem with scripts and interactive modes of using programming languages is that we're spending most of our time writing code for computers to read – which doesn't necessarily imply that our code is easy for **humans** to read.

There are two solutions to this problem, and I encourage you to make liberal use of both of them (together).

3.3.1 Code Comments

A **comment** is a part of computer code which is intended only for people to read. It is not evaluated or run by the computing language.

To "comment out" a single line of code in R or python, put a `#` (pound sign/hashtag) just before the part of the code you do not want to be evaluated.

Example: Comments

3.3.1.0.1 R

```
2 + 2 + 3
```

```
[1] 7
```

```
2 + 2 # + 3
```

```
[1] 4
```

```
# This line is entirely commented out
```

3.3.1.0.2 Python

```
2 + 2 + 3
```

```
7
```

```
2 + 2 # + 3
# This line is entirely commented out
```

```
4
```

Many computing languages, such as Java, C/C++, and JavaScript have mechanisms to comment out an entire paragraph. Neither R nor Python has so-called "block comments" - instead, you can use keyboard shortcuts in RStudio to comment out an entire chunk of code (or text) using Ctrl/Cmd-Shift-C.

3.3.2 Literate Programming - Notebooks and more!

While code comments add human-readable text to code, scripts with comments are still primarily formatted for the computer's convenience. However, most of the time spent on any given document is spent by people, not by computers. We often

write parallel documents - user manuals, academic papers, tutorials, etc. which explain the purpose of our code and how to use it, but this can get clumsy over time, and requires updating multiple documents (sometimes in multiple places), which often leads to the documentation getting out-of-sync from the code.

To solve this problem, Donald Knuth invented the concept of **literate programming**: interspersing text and code in the *same document* using structured text to indicate which lines are code and which lines are intended for human consumption.

This textbook is written using a literate format - quarto markdown - which allows me to include code chunks in R, python, and other languages, alongside the text, pictures, and other formatting necessary to create a textbook.

3.3.2.1 Quarto

One type of literate programming document is a **quarto markdown** document.

We will use quarto markdown documents for most of the components of this class because they allow you to answer assignment questions, write reports with figures and tables generated from data, and provide code all in the same file.

While literate documents aren't ideal for jobs where a computer is doing things unobserved (such as pulling data from a web page every hour), they are extremely useful in situations where it is desireable to have both code and an explanation of what the code is doing and what the results of that code are in the same document.

> Your turn: Quarto Markdown
>
> In RStudio, create a new quarto markdown document: File > New File > Quarto Document. Give your document a title and an author, and select HTML as the output.
> Copy the following text into your document and hit the

"Render" button at the top of the file.

````
This defines an R code chunk. The results will be included in the compiled HTML file.

```{r}
2 + 2
```

This defines a python code chunk. The results will be included in the compiled HTML file.

```{python}
2 + 2
```

# This is a header

## This is a subheader

I can add paragraphs of text, as well as other structured text such as lists:

1. First thing
2. Second thing
  - nested list
  - nested list item 2
3. Third thing

I can even include images and [links](https://www.oldest.org/entertainment/memes/)

![Goodwin's law is almost as old as the internet itself.](https://www.oldest.org/wp-conten
````

Markdown is a format designed to be readable and to allow document creators to focus on content rather than style.

> A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. – John Gruber

You can read more about pandoc markdown (and quarto markdown, which is a specific type of pandoc markdown) here [3].

Markdown documents are **compiled** into their final form (usually, HTML, PDF, Docx) in multiple stages: 1. All code chunks are run and the results are saved and inserted into the markdown document.
Rmd/qmd -> md

2. The markdown document is converted into its final format using **pandoc**, a program that is designed to ensure you can generate almost any document format. This may involve conversion to an intermediate file (e.g. .tex files for PDF documents).

An error in your code will likely cause a failure at stage 1 of the process. An error in the formatting of your document, or missing pictures, and miscellaneous other problems may cause errors in stage 2.

> **i** History
>
> Quarto markdown is the newest version of a long history of literate document writing in R. A previous version, Rmarkdown, had to be compiled using R; quarto can be compiled using R or python or the terminal directly.
> Prior to Rmarkdown, the R community used `knitr` and `Sweave` to integrate R code with LaTeX documents (another type of markup document that has a steep learning curve and is harder to read).

3.3.2.2 Jupyter

Where quarto comes primarily out of the R community and those who are agnostic whether R or Python is preferable for data science related computing, Jupyter is essentially an equivalent notebook technology that comes from the python side of the world.

Quarto supports using the jupyter engine for chunk compilation, but jupyter notebooks have some (rather technical) features that make them less desirable for an introductory computing class [4].

Learn More about Notebooks

There are some excellent opinions surrounding the use of notebooks in data analysis:

- Why I Don't Like Notebooks" by Joel Grus at JupyterCon 2018
- The First Notebook War by Yihui Xie (response to Joel's talk).

Yihui Xie is the person responsible for `knitr` and `Rmarkdown` and was involved in the development of `quarto`.

3.4 References

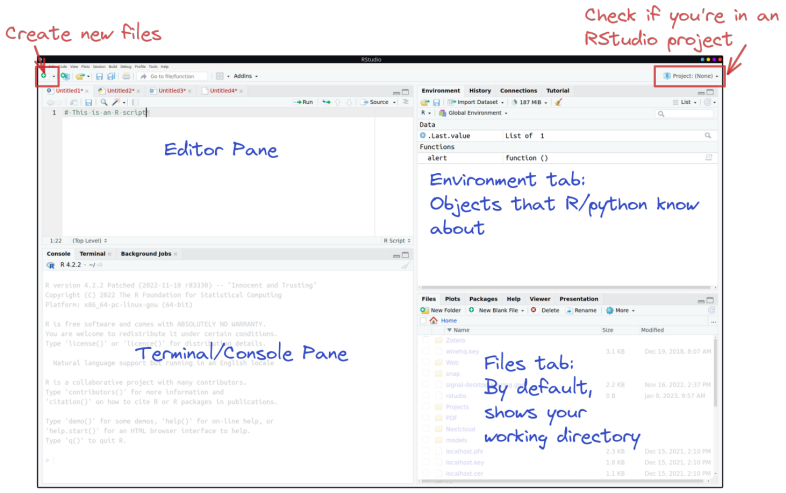
4 RStudio's Interface

Here, I'll provide a set of annotated screenshots highlighting many of the features of RStudio's IDE (Integrated Development Environment) which will be useful as you work through this textbook. You shouldn't expect to remember all of this right now, but I'm providing it in the hopes that you'll be able to come back to it when future instructions like "Click the render button at the top of the text editor window" don't make sense or match what you're seeing.

4.1 Objectives

- Locate different panes of RStudio
- Use cues such as buttons and icons to identify what type of file is open and what language is being interpreted

4.2 Overview



An RStudio window is by default divided into 4 panes, each of which may contain several tabs. You can reconfigure the locations of these tabs based on your preferences by selecting the toolbar button with 4 squares (just left of the Addins dropdown menu).

In the default configuration, - The top left is the editor pane, where you will write code and other content. - The bottom left is the console pane, which contains your R/python interactive consoles as well as a system terminal and location for checking the status of background jobs. - The top right contains the environment and history tabs (among others) - The top left contains the files and help tabs (among others)

You do not need to know what all of these tabs do right now. For the moment, it's enough to get a sense of the basics - where to write code (top left), where to look for results (bottom left), where to get help (bottom right), and where to monitor what R/python are doing (top right).

4.3 The Editor/File Pane (Top Left)

The buttons and layout within this pane change based on the type of file you have open.

4.3.0.1 R script

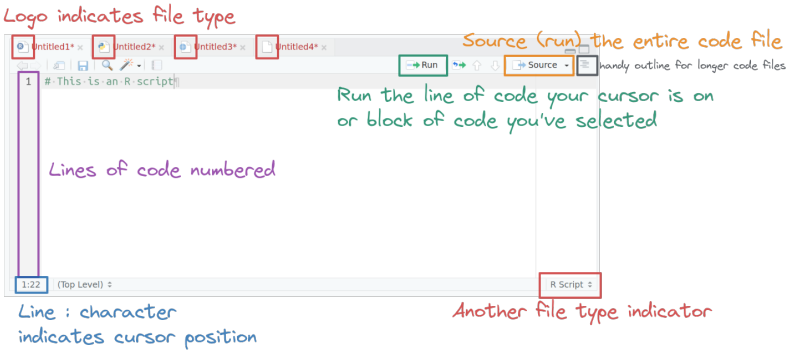


Figure 4.1: The logo on the script file indicates the file type. When an R file is open, there are Run and Source buttons on the top which allow you to run selected lines of code (Run) or source (run) the entire file. Code line numbers are provided on the left (this is a handy way to see where in the code the errors occur), and you can see line:character numbers at the bottom left. At the bottom right, there is another indicator of what type of file Rstudio thinks this is.

4.3.0.2 Python script

4.3.0.3 Quarto markdown

4.3.0.4 Text file

4.4 The Console Pane (Bottom Left)

Let's compare what the console pane looks like when we run a line of R code compared to a line of python code. The differences will help you figure out whether you need to exit out of Python to run R code and may help you debug some errors.

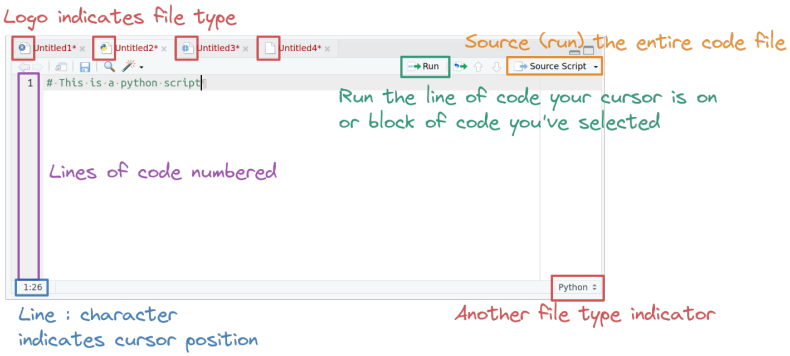


Figure 4.2: The logo on the script file indicates the file type. When a python file is open, there are Run and Source buttons on the top which allow you to run selected lines of code (Run) or source (run) the entire file. Code line numbers are provided on the left (this is a handy way to see where in the code the errors occur), and you can see line:character numbers at the bottom left. At the bottom right, there is another indicator of what type of file Rstudio thinks this is.

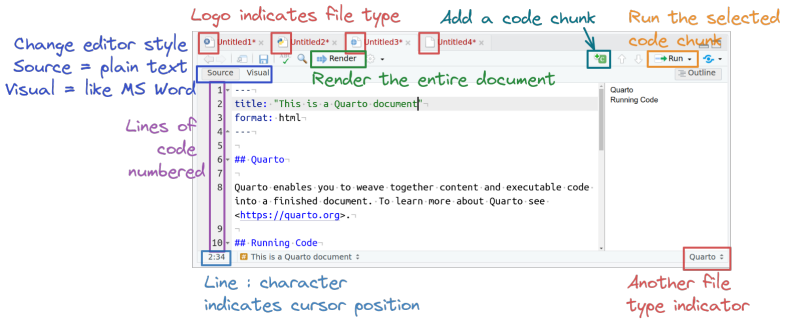


Figure 4.3: The logo on the script file indicates the file type. When a quarto markdown file is open, there is a render button at the top which allows you to compile the file to see its "pretty", non-markup form. In the same toolbar, there are buttons to add a code chunk as well as to run a selcted line of code or chunk of code. You can toggle between source (shown) and visual mode to see a more word-like rendering of the quarto markdown file. Code line numbers are provided on the left (this is a handy way to see where in the code the errors occur), and you can see line:character numbers at the bottom left. At the bottom right, there is another indicator of what type of file Rstudio thinks this is.

Figure 4.4: The logo on the text file indicates the file type. When a text file (or other unknown file extension) is open, there are very few buttons in the editor window. Code line numbers are provided on the left (this is a handy way to see where in the code the errors occur), and you can see line:character numbers at the bottom left. At the bottom right, there is another indicator of what type of file Rstudio thinks this is.

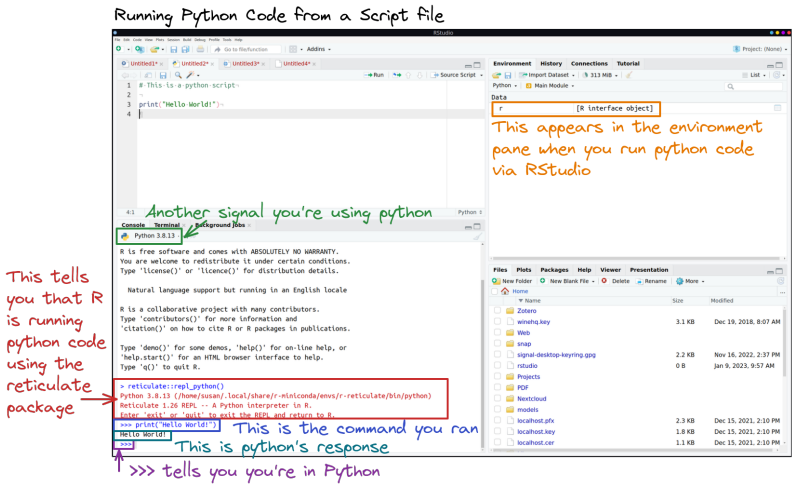


Figure 4.5: When running python code from a script file, the console will show you that you are running in python by the logo at the top of the console pane. You will initially see lines indicating that you're running R, and then you'll see the lines highlighted in red which show R running the code in python – this is what converts the console from R to python. The command you ran will appear after >>>, and the results will appear immediately below. A >>> waits for a new command - to get back to R, you will need to type exit (as instructed by the red text). In the environment pane, you cna see another indicator that you're viewing the python environment, with an object named 'r' that will allow you to move data back and forth between the two languages if you want to do so.

4.4.0.1 Python

4.4.0.2 R

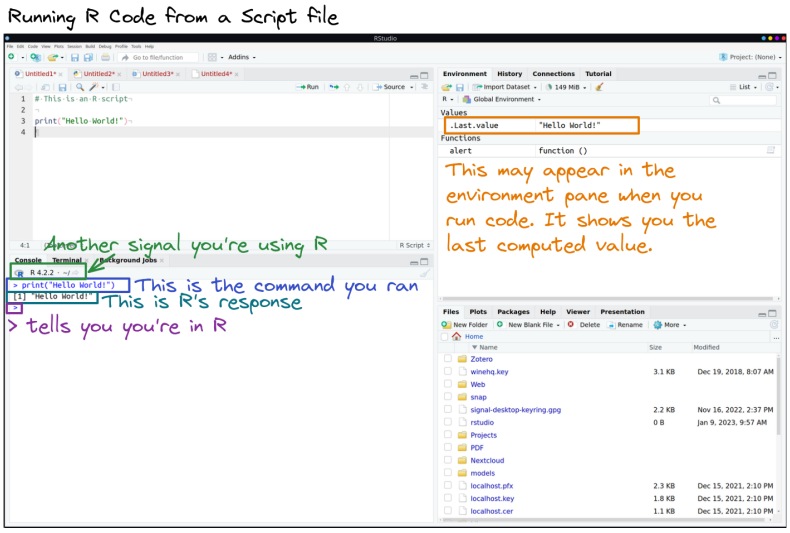


Figure 4.6: When running R code from a script file, the console will show you that you are running in R by the logo at the top of the console pane. You will initially see lines indicating that you're running R (they're missing here because this isn't the first command I ran in this session). The command you ran will appear after `>`, and the results will appear immediately below, with boxed numbers in front of each sequential line. A `>` waits for a new command . In the environment pane, you may see a new value pop up named `.Last.value` - this is part of user settings and you can stop it from appearing if you want to.

4.5 The Top Right Pane

This pane contains a set of tabs that change based on your project and what you have enabled. If you're using git with an Rstudio project, then this tab will show your git repository. If

you're working with an Rstudio project that has multiple files, such as a book or a website, then the pane will also have a Build tab that will build all of your project files.

For now, though, let's assume you're not in an Rstudio project and you just want to know what the heck an Environment pane (or any of the other tabs in here by default) is. We're going to focus on two of the tabs that are the most relevant to you right now: Environment, and History.

4.5.1 Environment tab

The Environment tab shows you any objects which are defined in memory in whatever language you're currently using (as long as it's R or python). You'll see headers like "Data", "Values", and "Functions" within this table, and two columns - the name of the thing, and the value of the thing (if it's a complicated object, you'll see what type of object it is and possibly how long it is).

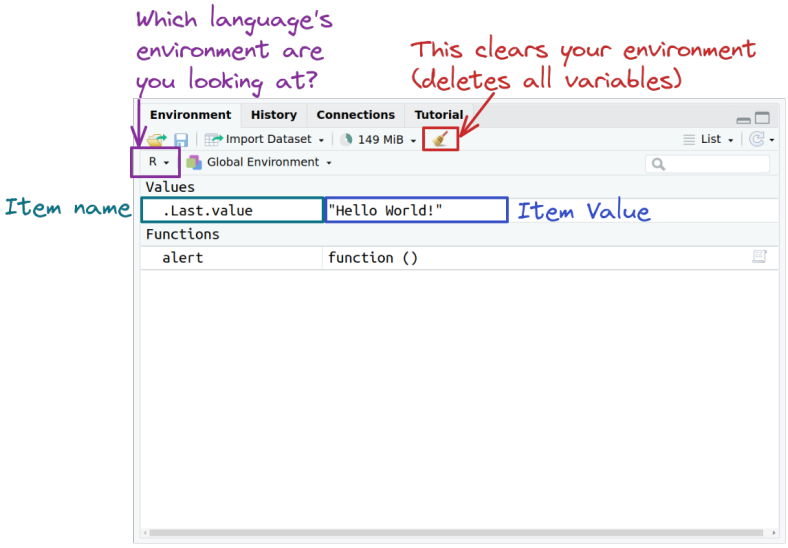


Figure 4.7: The environment tab shows you all of the objects in memory that the language you're working in knows about.

If you're working in both R and python, you can toggle which language's environment you're looking at using the language drop down button on the far left side.

4.5.2 History tab

Another useful tab in this pane is the History tab, which shows you a running list of every command you've ever run. While I strongly encourage you to write your code in a text file in the editor pane, sometimes you deleted a line of code accidentally and want to get it back... and the history tab has you covered (unless you've cleared the history out).

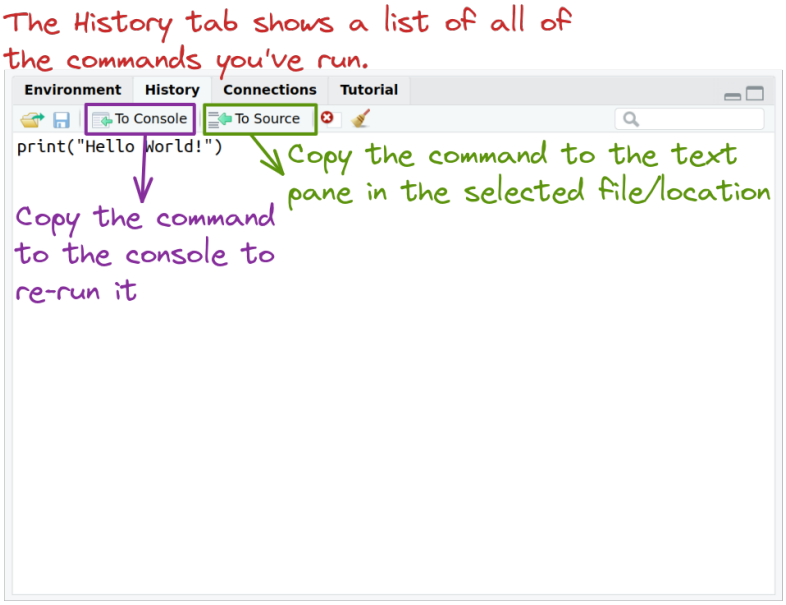


Figure 4.8: The history tab shows you a list of all commands you've run and allows you to send them to the console or to source (the text editor).

4.6 The Bottom Right Pane

This pane also contains a mishmash of tabs that have various uses. Here, we'll focus on 3: Files, Packages, and Help. But

first, to quickly summarize the remaining tabs, the Plots tab shows any plots you've generated (which we haven't done yet), and the Viewer/Presentation tabs show you compiled documents (markdown), interactive graphics, and presentations.

4.6.1 Files tab

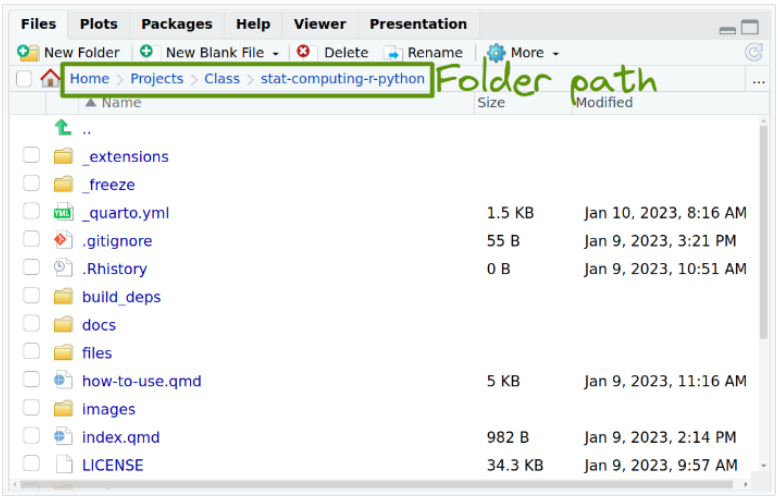


Figure 4.9: The files tab shows you the files in your current working directory (by default), though you can navigate through it and find other files as necessary. If you want to return to your working directory, there's a button for that in the "More" menu. One of the most important pieces of information in this pane is your path - you can construct the file path by using ~/ for home, and then for each folder, adding a slash between. The path to the folder we're looking at here is thus `~/Projects/Class/stat-computing-r-python/`.

4.6.2 Packages tab

The packages tab isn't quite relevant *yet*, but it will be soon. R and python both work off of packages - extensions to the default language that make it easier to accomplish certain tasks, like

reading data from Excel files or drawing pretty charts. This tab shows all of the R packages you have installed on your machine, and which ones are currently loaded.

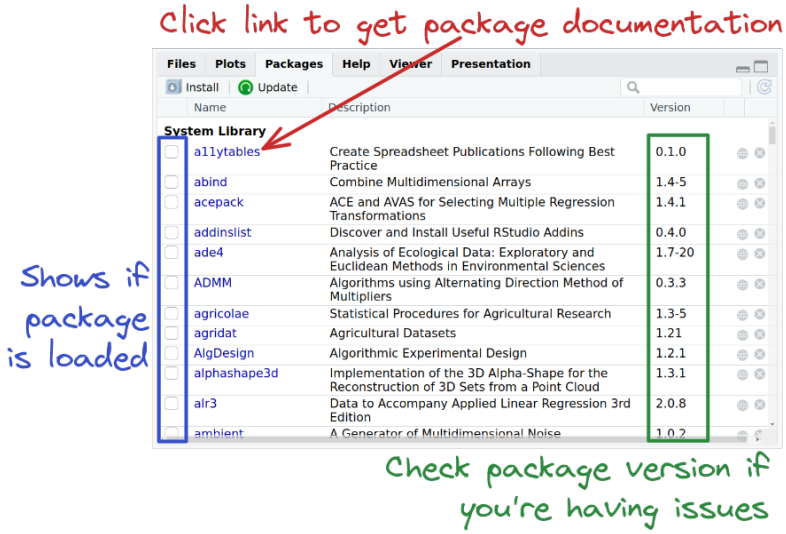


Figure 4.10: You can get important information from the packages tab, like what packages are loaded, easy access to documentation for each package, and what version of the package is installed.

Unfortunately, the packages tab doesn't cover python packages yet.

4.6.3 Help tab

The help tab is a wonderful way to get help with how to use an R or python function.

By default, you can search for an R function name in the search window, and documentation for matching functions will appear in the main part of the pane. To get help with python functions, you need to (in the python console) use `?<function name`, so I would type in at the `>>>` prompt `?print` to get the equivalent python help file.

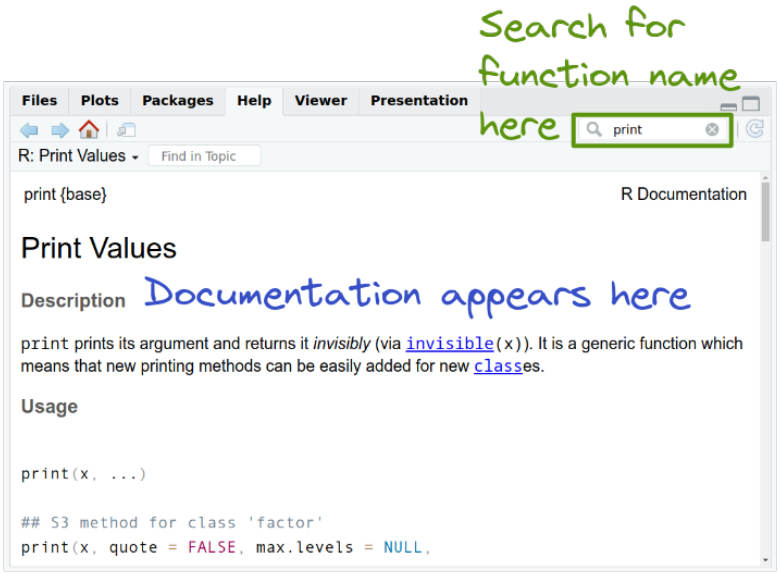


Figure 4.11: The help tab makes it easy to get access to function documentation within Rstudio, so you don't have to switch windows.

5 Version Control with Git

There is an entire textbook on how to use git and GitHub with R, Happy Git and Github for the UseR [5]. This chapter will liberally use chunks of that textbook, and rather than reproduce them here, I will simply link to the relevant sections.

5.1 Objectives

- Install git
- Create a github account
- Understand why version control is useful and what problems it can solve
- Understand the distinction between git and github, and what each is used for
- Use version control to track changes to a document (git add, commit, push, pull)

5.2 Installation

1. Install git using the instructions here.

2. Consult the troubleshooting guide if you have issues.
3. If 1-2 fail, seek help in office hours.

> ⚠ Mac Warning
>
> With each version upgrade, you may find that git breaks. To fix it, you will have to reinstall Mac command line tools. Once you do this, git will start working again. See [6] for more information.

5.2.1 Optional: Install a git client

Instructions

I don't personally use a git client other than RStudio, but you may prefer to have a client that allows you to use a point-and-click interface. It's up to you.

5.3 What is Version Control ?

> **Note**
>
> Most of this section is either heavily inspired by Happy Git and Github for the UseR [5] or directly links to that book.

Git is a **version control system** - a structured way for tracking changes to files over the course of a project that may also make it easy to have multiple people working on the same files at the same time.

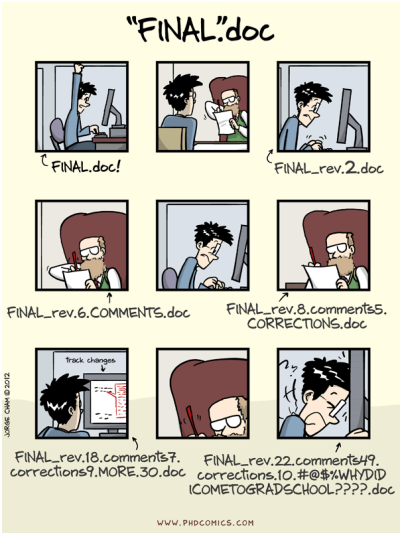


Figure 5.1: Version control is the answer to this file naming problem. Image Source "Piled Higher and Deeper" by Jorge Cham www.phdcomics.com

Git manages a collection of files in a structured way - rather like "track changes" in Microsoft Word or version history in Dropbox, but much more powerful.

If you are working alone, you will benefit from adopting version control because it will remove the need to add _final.R to the end of your file names. However, most of us work in collaboration with other people (or will have to work with others eventually), so one of the goals of this program is to teach you how to use git because it is a useful tool that will make you a better collaborator.

In data science programming, we use git for a similar, but slightly different purpose. We use it to keep track of changes not only to code files, but to data files, figures, reports, and other essential bits of information.

Git itself is nice enough, but where git really becomes amazing is when you combine it with GitHub - an online service that makes it easy to use git across many computers, share information with collaborators, publish to the web, and more. Git is great, but GitHub is … essential. In this class, we'll be using both git and github, and your homework will be managed with GitHub Classroom.

5.3.1 Git Basics

Git tracks changes to each file that it is told to monitor, and as the files change, you provide short labels describing what the changes were and why they exist (called "**commits**"). The log of these changes (along with the file history) is called your **git commit history**.

When writing papers, this means you can cut material out freely, so long as the paper is being tracked by git - you can always go back and get that paragraph you cut out if you need to. You also don't have to rename files - you can confidently save over your old files, so long as you remember to commit frequently.

Figure 5.2: If that doesn't fix it, git.txt contains the phone number of a friend of mine who understands git. Just wait through a few minutes of 'It's really pretty simple, just think of branches as…' and eventually you'll learn the commands that will fix everything. Image by Randall Munroe (XKCD) CC-A-NC-2.5.

! Essential Reading: Git

The git material in this chapter is just going to link directly to the book "Happy Git with R" by Jenny Bryan. It's amazing, amusing, and generally well written. I'm not going to try to do better.
Go read Chapter 1, if you haven't already.

Now that you have a general idea of how git works and why we might use it, let's talk a bit about GitHub.

5.3.2 GitHub: Git on the Web

! Set up a GitHub Account Now

Instructions for setting up a GitHub account.
Be sure you remember your signup email, username, and password - you will need them later.

Git is a program that runs on your machine and keeps track of changes to files that you tell it to monitor. GitHub is a website that hosts people's git repositories. You can use git without GitHub, but you can't use GitHub without git.

i Git and Github: Slightly crude (but memorable) analogy

Git is to GitHub what Porn is to PornHub. Specifically, GitHub hosts git repositories publicly, while PornHub hosts porn publicly. But it would be silly to equate porn and PornHub, and it's similarly silly to think of GitHub as the only place you can use git repositories.

If you want, you can hook Git up to GitHub, and make a copy of your local git repository that lives in the cloud. Then, if you configure things correctly, your local repository will talk to GitHub without too much trouble. Using Github with Git allows you to **easily make a cloud backup of your important code**, so that even if your computer suddenly catches on fire, all of your important code files exist somewhere else.

Remember: any data you don't have in 3 different places is data you don't care about.[1]

5.4 Using Version Control (with RStudio)

The first skill you need to actually practice in this class is using version control. By using version control from the very beginning, you will learn better habits for programming, but you'll also get access to a platform for collaboration, hosting your work online, keeping track of features and necessary changes, and more.

So, what does your typical git/GitHub workflow look like? I'll go through this in (roughly) chronological order. This is based off of a relatively high-level understanding of git - I do not have any idea how it works under the hood, but I'm pretty comfortable with the clone/push/pull/commit/add workflows, and I've used a few of the more complicated features (branches, pull requests) on occasion.

5.4.1 Introduce yourself to git and set up SSH authentication

You need to tell git what your name and email address are, because every "commit" you make will be signed. This needs to be done once on each computer you're using.

Follow the instructions here, or run the lines below:

> **Note**
>
> The lines of code below use **interactive prompts**. Click the copy button in the upper right corner of the box below, and then paste the *whole thing* into the R console. You

[1] Yes, I'm aware that this sounds paranoid. It's been a very rare occasion that I've needed to restore something from another backup. You don't want to take chances. I knew a guy who had to retype his entire masters thesis from the printed out version the night before it was due because he had stored it on a network drive that was decommissioned. You don't want to be that person.

will see a line that says "Your full name:" - type your name into the console. Similarly, the next line will ask you for an email address.)

```
user_name <- readline(prompt = "Your full name: ")
user_email <- readline(prompt = "The address associated w your github account: ")

install.packages("usethis")
library(usethis)

use_git_config(user.name = user_name, user.email = user_email, scope = "user")

# Tell git to ignore all files that are OS-dependent and don't have useful data.
git_vaccinate()

# Create a ssh key if one doesn't already exist
if (!file.exists(git2r::ssh_path("id_rsa.pub"))) {
  # Create an ssh key (with no password - less secure, but simpler)
  system("ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa -q -N ''")
  # Find the ssh-agent that will keep track of the password
  system("eval $(ssh-agent -s)")
  # Add the key
  system("ssh-add ~/.ssh/id_rsa")
}
```

Then, in RStudio, go to Tools > Global Options > Git/SVN. View your public key, and copy it to the clipboard.

Then, proceed to github. Make sure you're signed into GitHub. Click on your profile pic in upper right corner and go Settings, then SSH and GPG keys. Click "New SSH key". Paste your public key in the "Key" box. Give it an informative title. For example, you might use `2022-laptop` to record the year and computer. Click "Add SSH key".

5.4.2 Create a Repository

Repositories are single-project containers. You may have code, documentation, data, TODO lists, and more associated

with a project. If you combine a git repository with an RStudio project, you get a very powerful combination that will make your life much easier, allowing you to focus on writing code instead of figuring out where all of your files are for each different project you start.

To create a repository, you can start with your local computer first, or you can start with the online repository first.

> **!** Important
>
> Both methods are relatively simple, but the options you choose depend on which method you're using, so be careful not to get them confused.

5.4.2.1 Local repository first

Let's suppose you already have a folder on your machine named `hello-world-1` (you may want to create this folder now). You've created a starter document, say, a text file named README with "hello world" written in it.

If you want, you can use the following R code to set this up:

```
dir <- "./hello-world-1"
if (!dir.exists(dir)) {
  dir.create(dir)
}
file <- file.path(dir, "README")
if (!file.exists(file)) {
  writeLines("hello world", con = file)
}
```

To create a local git repository, we can go to the terminal (in Mac/Linux) or the git bash shell (in Windows), navigate to our repository folder (not shown, will be different on each computer), and type in

```
git init
```

Alternately, if you prefer a GUI (graphical user interface) approach, that will work too:

1. Open Rstudio
2. Project (upper right corner) -> New Project -> Existing Directory. Navigate to the directory.
3. (In your new project) Tools -> Project options -> Git/SVN -> select git from the dropdown, initialize new repository. RStudio will need to restart.
4. Navigate to your new Git tab on the top right.

The next step is to add our file to the repository.

Using the command line, you can type in `git add README` (this tells git to track the file) and then commit your changes (enter them into the record) using `git commit -m "Add readme file"`.

Using the GUI, you navigate to the git pane, check the box next to the README file, click the Commit button, write a message ("Add readme file"), and click the commit button.

The final step is to create a corresponding repository on GitHub. Navigate to your GitHub profile and make sure you're logged in. Create a new repository using the "New" button. Name your repository whatever you want, fill in the description if you want (this can help you later, if you forget what exactly a certain repo was *for*), and DO NOT add a README, license file, or anything else (if you do, you will have a bad time).

You'll be taken to your empty repository, and git will provide you the lines to paste into your git shell (or terminal) – you can access this within RStudio, as shown below. Paste those lines in, and you'll be good to go.

5.4.2.2 GitHub repository first

In the GitHub-first method, you'll create a repository in GitHub and then clone it to your local machine (clone = create an exact copy locally).

GUI method:

1. Log into GitHub and create a new repository
2. Initialize your repository with a README
3. Copy the repository location by clicking on the "Code" button on the repo homepage
4. Open RStudio -> Project -> New Project -> From version control. Paste your repository URL into the box. Hit enter.
5. Make a change to the README file
6. Click commit, then push your changes
7. Check that the remote repository (Github) updated

Command line method:

1. Log into GitHub and create a new repository
2. Initialize your repository with a README
3. Copy the repository location by clicking on the "Code" button on the repo homepage
4. Navigate to the location you want your repository to live on your machine.
5. Clone the repository by using the git shell or terminal: `git clone <your repo url here>`. In my case, this looks like `git clone git@github.com:stat850-unl/hello-world-2.git`
6. Make a change to your README file and save the change
7. Commit your changes: `git commit -a -m "change readme"` (-a = all, that is, any changed file git is already tracking).
8. Push your changes to the remote (GitHub) repository and check that the repo has updated: `git push`

5.4.3 Adding files

`git add` tells git that you want it to track a particular file.

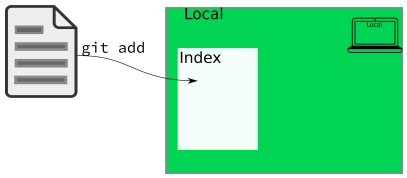


Figure 5.3: git add diagram: add tells git to add the file to the index of files git monitors.

You don't need to understand exactly what git is doing on the backend, but it is important to know that the actual **contents** of the file aren't logged by `git add` - you have to **commit** your changes for the contents to change. `git add` deals solely with the index of files that git "knows about", and what it thinks belongs in each commit.

If you use the RStudio GUI for your git interface, you generally won't have to do much with git add; it's (sort-of, kind-of) equivalent to clicking the check box.

5.4.3.1 What files should I add to git?

Git is built for tracking text files. It will (begrudgingly) deal with small binary files (e.g. images, PDFs) without complaining too much, but it is NOT meant for storing large files, and GitHub will not allow you to push anything that has a file larger than 100MB[2]. Larger files can be handled with git-lfs (large file storage), but storing large files online is not something you can get for free.

In general, you should **only add a file to git if you created it by hand**. If you compiled the result, that should not be in the git repository under normal conditions (there are exceptions to this rule – this book is hosted on GitHub, which means I've pushed the compiled book to the GitHub repository).

You should also be cautious about adding files like .Rprog, .directory, .DS_Store, etc. These files are used by your operating system or by RStudio, and pushing them may cause problems for your collaborators (if you're collaborating). Tracking changes to these files also doesn't really do much good.

I **highly** recommend that you make a point to only add and commit files which you consciously want to track.

[2]Yes, I'm seriously pushing it with this book; several of the datasets are ~30 MB

5.4.4 Staging your changes

In RStudio, when you check a box next to the file name in the git tab, you are effectively adding the file (if it is not already added) AND staging all of the changes you've made to the file. In practice, `git add` will both add and stage all of the changes to any given file, but it is also useful in some cases to stage only certain lines from a file.

More formally, **staging** is saying "I'd like these changes to be added to the current version, I think". Before you **commit** your changes, you have to first **stage** them. You can think of this like going to the grocery store: you have items in your cart, but you can put them back at any point before checkout. Staging changes is like adding items to your cart; committing those changes is like checking out.

Individually staging lines of a file is most useful in situations where you've made changes which should be part of multiple commits. To stage individual lines of a file, you can use `git add -i` at the command line, or you can attempt to use RStudio's "stage selection" interface. Both will work, though git can't always separate changes quite as finely as you might want (and as a result, RStudio's interface sometimes seems unresponsive, even though the underlying issue is with what git can do).

5.4.5 Committing your changes

A git **commit** is the equivalent of a log entry - it tells git to record the state of the file, along with a message about what that state means. On the back end, git will save a copy of the file in its current state to its cache.

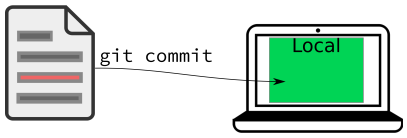


Figure 5.4: Here, we commit the red line as a change to our file.

In general, you want your commit message to be relatively short, but also informative. The best way to do this is to commit **small** blocks of changes. Work to commit every time you've accomplished a small task. This will do two things:

1. You'll have small, bite-sized changes that are briefly described to serve as a record of what you've done (and what still needs doing)
2. When you mess up (or end up in a merge conflict) you will have a much easier time pinpointing the spot where things went bad, what code was there before, and (because you have nice, descriptive commit messages) how the error occurred.

5.4.6 Pushing and Pulling

When you're working alone, you generally won't need to worry about having to update your local copy of the repository (unless you're using multiple machines). However, statistics is collaborative, and one of the most powerful parts of git is that you can use it to keep track of changes when multiple people are working on the same document.

If you are working collaboratively and you and your collaborator are working on the same file, git will be able to resolve the change you make SO LONG AS YOU'RE NOT EDITING THE SAME LINE. Git works based on lines of text - it detects when there is a change in any line of a text document.

For this reason, I find it makes my life easier to put each sentence on a separate line, so that I can tweak things with fewer merge conflicts. Merge conflicts aren't a huge deal, but they slow the workflow down, and are best avoided where possible.

Pulling describes the process of updating your local copy of the repository (the copy on your computer) with the files that are "in the cloud" (on GitHub). `git pull` (or using the Pull button in RStudio) will perform this update for you. If you are working with collaborators in real time, it is good practice to pull, commit, and push often, because this vastly reduces the merge conflict potential (and the scope of any conflicts that do pop up).

Pushing describes the process of updating the copy of the repository on another machine (e.g. on GitHub) so that it has the most recent changes you've made to your machine.

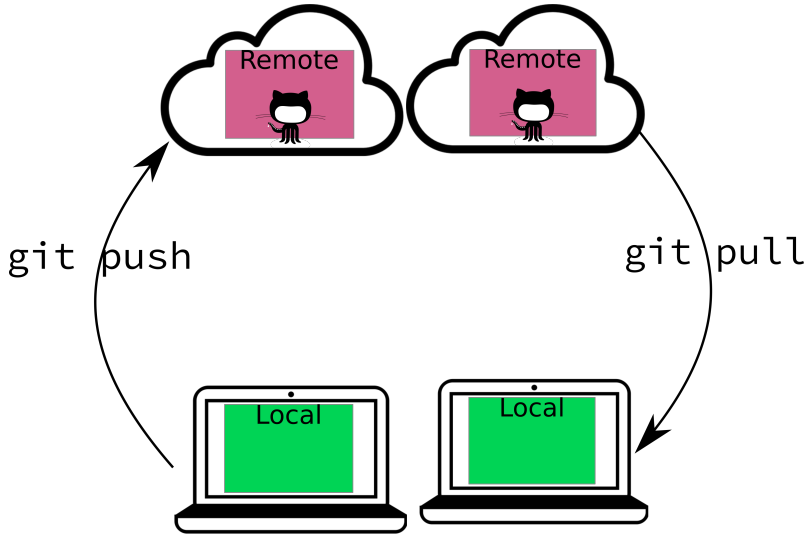


Figure 5.5: git push copies the version of the project on your computer to GitHub

Figure 5.6: git pull copies the version of the project on GitHub to your computer

Figure 5.7: Git push and git pull are used to sync your computer with the remote repository (usually hosted on GitHub)

In general, your workflow will be

1. Clone the project or create a new repository
2. Make some changes
3. Stage the changes with git add
4. Commit the changes with git commit
5. Pull any changes from the remote repository
6. Resolve any merge conflicts
7. Push the changes (and merged files) with git push

If you're working alone, steps 5 and 6 are not likely to be necessary, but it is good practice to just pull before you push

anyways.

5.5 References

Part II

Part II: General Programming

In this portion of the textbook, we'll talk about the basics of programming in a general sense (that is, we're not yet focusing on programming *with data*).

Before we start in on the hard stuff, we'll quickly go through what programming is and what the vocabulary of a programming language looks like (this section).

Chapter 6 will discuss the basics: variable types, how to assign variables, and how to convert between simple variable types.

Chapter 7 will discuss how to use built-in and package functions to make R and python more powerful. After this section, you should be able to use R or Python as a calculator.

Chapter 8 will discuss the use of vectors and matrices in R and Python. Along the way, you'll get a quick refresher in mathematical logic - the use of And, Or, and Not.

Chapter 9 will discuss control structures - ways to change the flow of a program based on variable values and operating condition. This will include discussions of if-statements and different types of loops.

Chapter 10 will discuss writing your own functions.

Once we've covered these topics, we should be ready to focus on programming with, for, and on data.

What is Programming?

> Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the universe trying to produce bigger and better idiots. So far, the universe is winning. - Rick Cook

Programming is the art of solving a problem by developing a sequence of steps that make up a solution, and then very carefully communicating those steps to the computer. To program, you need to know how to

- break a problem down into smaller, easily solvable problems
- solve the small problems
- communicate the solution to a computer using a programming language

In this book, we'll be using both R and Python, and we'll be using these languages to solve problems that are related to working with data. At first, we'll start with smaller, simpler problems that don't involve data, but by the end, you will hopefully be able to solve some statistical problems using one or both languages.

It will be hard at first - you have to learn the vocabulary in both languages in order to be able to put commands into logical "sentences". The problem solving skills are the same for all programming languages, though, and while those are harder to learn, they'll last you a lifetime.

Just as you wouldn't expect to learn French or Mandarin fluently after taking a single class, you cannot expect to be fluent in R or python once you've worked through this book. Fluency takes years of work and practice, and lots of mistakes along the way. You *cannot* learn a language (programming or otherwise) if you're worried about making mistakes. Take a minute and put those concerns away somewhere, take a deep breath, and remember the Magic School Bus Motto:

Figure 5.8: For those who don't know, the Magic School Bus is a PBS series that aired in the 1990s and was brought back by Netflix in 2017. It taught kids about different principles of science and the natural world.

Programming Vocabulary: Hello World

I particularly like the way that Python for Everybody [7] explains vocabulary:

> Unlike human languages, the Python vocabulary is actually pretty small. We call this "vocabulary" the "reserved words". These are words that have very special meaning to Python. When Python sees these words in a Python program, they have one and only one meaning to Python. Later as you write programs you will make up your own words that have meaning to you called variables. You will have great latitude in choosing your names for your variables, but you cannot use any of Python's reserved words as a name for a variable.
>
> When we train a dog, we use special words like "sit", "stay", and "fetch". When you talk to a dog and don't use any of the reserved words, they just look at you with a quizzical look on their face until you say a reserved word. For example, if you say, "I wish more people would walk to improve their overall health", what most dogs likely hear is, "blah blah blah walk blah blah blah blah." That is because "walk" is a reserved word in dog language. Many might suggest that the language between humans and cats has no reserved words.
>
> The reserved words in the language where humans talk to Python include the following:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

That is it, and unlike a dog, Python is already completely trained. When you say 'try', Python will try every time you say it without fail.

We will learn these reserved words and how they are used in good time, but for now we will focus on the Python equivalent of "speak" (in human-to-dog language). The nice thing about telling Python to speak is that we can even tell it what to say by giving it a message in quotes:

```
print('Hello world!')
## Hello world!
```

And we have even written our first syntactically correct Python sentence. Our sentence starts with the function print followed by a string of text of our choosing enclosed in single quotes. The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

R has a slightly smaller set of reserved words:

```
if          else     repeat      while
for         in       next        break
TRUE        FALSE    NULL        Inf
NA_integer_ NA_real_ NA_complex_ NA_character_
NaN         NA       function    ...
```

In R, the "Hello World" program looks exactly the same as it does in python.

```
print('Hello world!')
## [1] "Hello world!"
```

In many situations, R and python will be similar because both languages are based on C. R has a more complicated history [8], because it is also similar to Lisp, but both languages are still very similar to C and run C or C++ code in the background.

Getting help

In both R and python, you can access help with a `?` - the order is just slightly different.

Suppose we want to get help on a `for` loop in either language.

In R, we can run this line of code to get help on `for` loops.

```
?`for`
```

Because `for` is a reserved word in R, we have to use backticks (the key above the `TAB` key) to surround the word `for` so that R knows we're talking about the function itself. Most other function help can be accessed using `?function_name`. The backtick trick also works for functions that don't start with letters, like `+`.

In python, we use `for?` to access the same information.

```
for? # help printed in the terminal
?for # help printed in the help pane
```

(You will have to run this in interactive mode for it to work in either language)

w3schools has an excellent python help page that may be useful as well. Searching for help using google also works well, particularly if you know what sites are likely to be helpful, like w3schools and stackoverflow. A similar set of pages exists for R help on basic functions

> **Learn More**
>
> A nice explanation of the difference between an interpreter and a compiler. Both Python and R are interpreted languages that are *compiled* from lower-level languages like C.

References

6 Variables and Basic Data Types

6.1 Objectives

- Know the basic data types and what their restrictions are
- Know how to test to see if a variable is a given data type
- Understand the basics of implicit and explicit type conversion
- Write code that assigns values to variables

6.2 Basic Definitions

For a general overview, [9] is an excellent introduction to data types:

Let's start this section with some basic vocabulary.

- a **value** is a basic unit of stuff that a program works with, like `1`, `2`, `"Hello, World"`, and so on.
- values have **types** - `2` is an integer, `"Hello, World"` is a string (it contains a "string" of letters). Strings are in quotation marks to let us know that they are not variable names.

In most programming languages (including R and python), there are some very basic data types:

- **logical** or **boolean** - FALSE/TRUE or 0/1 values. Sometimes, boolean is shortened to **bool**
- **integer** - whole numbers (positive or negative)
- **double** or **float** or **numeric**- decimal numbers.

  - **float** is short for floating-point value.
  - **double** is a floating-point value with more precision ("double precision").[1]
  - R uses the name **numeric** to indicate a decimal value, regardless of precision.

- **character** or **string** - holds text, usually enclosed in quotes.

> **!** Capitalization matters!
>
> In R, boolean values are `TRUE` and `FALSE`, but in Python they are `True` and `False`. Capitalization matters a LOT. Other things matter too: if we try to write a million, we would write it `1000000` instead of `1,000,000` (in both languages). Commas are used for separating numbers, not for proper spacing and punctuation of numbers. This is a hard thing to get used to but very important – especially when we start reading in data.

6.3 Variables

Programming languages use **variables** - names that refer to values. Think of a variable as a container that holds something - instead of referring to the value, you can refer to the container and you will get whatever is stored inside.

6.3.1 Assignment

We **assign** variables values using the syntax `object_name <- value` (R) or `object_name = value` (python). You can read this as "object name gets value" in your head.

In R, `<-` is used for assigning a value to a variable. So `x <- "R is awesome"` is read "x gets 'R is awesome' " or "x is assigned

[1] This means that doubles take up more memory but can store more decimal places. You don't need to worry about this much in R, and only a little in Python, but in older and more precise languages such as C/C++/Java, the difference between floats and doubles can be important.

the value 'R is awesome' ". Technically, you can also use `=` to assign things to variables in R, but most style guides consider this to be poor programming practice, so seriously consider defaulting to `<-`.

In Python, `=` is used for assigning a value to a variable. This tends to be much easier to say out loud, but lacks any indication of directionality.

R

```r
message <- "So long and thanks for all the fish"
year <- 2025
the_answer <- 42L
earth_demolished <- FALSE
```

Python

```python
message = "So long and thanks for all the fish"
year = 2025
the_answer = 42
earth_demolished = False
```

> **Note**
>
> Note that in R, we assign variables values using the `<-` operator, where in Python, we assign variables values using the `=` operator. Technically, `=` will work for assignment in both languages, but `<-` is more common than `=` in R by convention.

We can then use the variables - do numerical computations, evaluate whether a proposition is true or false, and even manipulate the content of strings, all by referencing the variable by name.

6.3.2 Valid Names

> There are only two hard things in Computer Science: cache invalidation and naming things.
> – Phil Karlton

Object names must start with a letter and can only contain letters, numbers, `_`, and `.` in R. In Python, object names must start with a letter and can consist of letters, numbers, and `_` (that is, `.` is not a valid character in a Python variable name). While it is technically fine to use uppercase variable names in Python, it's recommended that you use lowercase names for variables (you'll see why later).

What happens if we try to create a variable name that isn't valid?

In both languages, starting a variable name with a number will get you an error message that lets you know that something isn't right - "unexpected symbol" in R and "invalid syntax" in python.

R

```
1st_thing <- "check your variable names!"
## Error: <text>:1:2: unexpected symbol
## 1: 1st_thing
##      ^
```

Python

```
1st_thing <- "check your variable names!"
```

Note: Run the above chunk in your python window - the book won't compile if I set it to evaluate . It generates an error of `SyntaxError: invalid syntax (<string>, line 1)`

```
second.thing <- "this isn't valid"
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'second' is no
```

In python, trying to have a `.` in a variable name gets a more interesting error: " is not defined". This is because in python, some objects have components and methods that can be accessed with `.`. We'll get into this more later, but there is a good reason for python's restriction about not using `.` in variable names.

Naming things is difficult! When you name variables, try to make the names descriptive - what does the variable hold? What are you going to do with it? The more (concise) information you can pack into your variable names, the more readable your code will be.

> **Learn More**
>
> Why is naming things hard? - Blog post by Neil Kakkar

There are a few different conventions for naming things that may be useful:

- `some_people_use_snake_case`, where words are separated by underscores
- `somePeopleUseCamelCase`, where words are appended but anything after the first word is capitalized (leading to words with humps like a camel).
- `some.people.use.periods` (in R, obviously this doesn't work in python)
- A few people mix conventions with `variables_thatLookLike.this` and they are almost universally hated

As long as you pick ONE naming convention and don't mix-and-match, you'll be fine. It will be easier to remember what you named your variables (or at least guess) and you'll have fewer moments where you have to go scrolling through your script file looking for a variable you named.

6.4 Types

You can use different functions to test whether a variable has a specific type as well:

R

```
is.logical(FALSE)
is.integer(2L) # by default, R treats all numbers as numeric/decimal values.
          # The L indicates that we're talking about an integer.
is.integer(2)
is.numeric(2)
is.character("Hello, programmer!")
is.function(print)
## [1] TRUE
## [1] TRUE
## [1] FALSE
## [1] TRUE
## [1] TRUE
## [1] TRUE
```

In R, you use `is.xxx` functions, where xxx is the name of the type in question.

Python

```
isinstance(False, bool)
isinstance(2, int)
isinstance(2, (int, float)) # Test for one of multiple types
isinstance(3.1415, float)
isinstance("This is python code", str)
## True
## True
## True
## True
## True
```

In python, test for types using the `isinstance` function with an argument containing one or more data types in a tuple (`(int, float)` is an example of a tuple - a static set of multiple values).

If we want to test for whether something is **callable** (can be used like a function), we have to get slightly more complicated:

```
callable(print)
## True
```

This is glossing over some much more technical information about differences between functions and classes (that we haven't covered) [10].

Example: Assignment and Testing Types

Character

```
x <- "R is awesome"
typeof(x)
## [1] "character"
is.character(x)
## [1] TRUE
is.logical(x)
## [1] FALSE
is.integer(x)
## [1] FALSE
is.double(x)
## [1] FALSE
```

```
x = "python is awesome"
type(x)
## <class 'str'>
isinstance(x, str)
## True
isinstance(x, bool)
## False
isinstance(x, int)
## False
isinstance(x, float)
## False
```

Logical

```r
x <- FALSE
typeof(x)
## [1] "logical"
is.character(x)
## [1] FALSE
is.logical(x)
## [1] TRUE
is.integer(x)
## [1] FALSE
is.double(x)
## [1] FALSE
```

In R, is possible to use the shorthand `F` and `T`, but be careful with this, because `F` and `T` are not reserved, and other information can be stored within them. See this discussion for pros and cons of using `F` and `T` as variables vs. shorthand for true and false. [a]

```python
x = False
type(x)
## <class 'bool'>
isinstance(x, str)
## False
isinstance(x, bool)
## True
isinstance(x, int)
## True
isinstance(x, float)
## False
```

Note that in python, boolean variables are also integers. If your goal is to test whether something is a T/F value, you may want to e.g. test whether its value is one of 0 or 1, rather than testing whether it is a boolean variable directly, since integers can also function directly as bools in Python.

Integer

```
x <- 2
typeof(x)
## [1] "double"
is.character(x)
## [1] FALSE
is.logical(x)
## [1] FALSE
is.integer(x)
## [1] FALSE
is.double(x)
## [1] TRUE
```

Wait, 2 is an integer, right?
2 is an integer, but in R, values are assumed to be doubles unless specified. So if we want R to treat 2 as an integer, we need to specify that it is an integer specifically.

```
x <- 2L # The L immediately after the 2 indicates that it is an integer.
typeof(x)
## [1] "integer"
is.character(x)
## [1] FALSE
is.logical(x)
## [1] FALSE
is.integer(x)
## [1] TRUE
is.double(x)
## [1] FALSE
is.numeric(x)
## [1] TRUE
```

```python
x = 2
type(x)
## <class 'int'>
isinstance(x, str)
## False
isinstance(x, bool)
## False
isinstance(x, int)
## True
isinstance(x, float)
## False
```

Double

```r
x <- 2.45
typeof(x)
## [1] "double"
is.character(x)
## [1] FALSE
is.logical(x)
## [1] FALSE
is.integer(x)
## [1] FALSE
is.double(x)
## [1] TRUE
is.numeric(x)
## [1] TRUE
```

```
x = 2.45
type(x)
## <class 'float'>
isinstance(x, str)
## False
isinstance(x, bool)
## False
isinstance(x, int)
## False
isinstance(x, float)
## True
```

Numeric

A fifth common "type"[b], `numeric` is really the union of two types: integer and double, and you may come across it when using `str()` or `mode()`, which are similar to `typeof()` but do not quite do the same thing.
The `numeric` category exists because when doing math, we can add an integer and a double, but adding an integer and a string is ... trickier. Testing for numeric variables guarantees that we'll be able to do math with those variables. `is.numeric()` and `as.numeric()` work as you would expect them to work.
The general case of this property of a language is called **implicit type conversion** - that is, R will implicitly (behind the scenes) convert your integer to a double and then add the other double, so that the result is unambiguously a double.

[a] There is also an R package dedicated to pure evil that will set F and T randomly on startup. Use this information wisely.
[b] `numeric` is not really a type, it's a mode. Run `?mode` for more information.

6.5 Type Conversions

Programming languages will generally work hard to seamlessly convert variables to different types. So, for instance,

R

```r
TRUE + 2
## [1] 3

2L + 3.1415
## [1] 5.1415

"abcd" + 3
## Error in "abcd" + 3: non-numeric argument to binary operator
```

Python

```python
True + 2
## 3
int(2) + 3.1415
## 5.141500000000001
"abcd" + 3
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: can only concatenat
```

This conversion doesn't always work - there's no clear way to make "abcd" into a number we could use in addition. So instead, R or python will issue an error. This error pops up frequently when something went wrong with data import and all of a sudden you just tried to take the mean of a set of string/character variables. Whoops.

When you want to, you can also use `as.xxx()` to make the type conversion **explicit**. So, the analogue of the code above, with explicit conversions would be:

R

```r
as.double(TRUE) + 2
## [1] 3

as.double(2L) + 3.1415
## [1] 5.1415
```

```r
as.numeric("abcd") + 3
## [1] NA
```

Python

```r
int(True) + 2
## 3
float(2) + 3.1415
## 5.141500000000001
float("abcd") + 3
## Error in py_call_impl(callable, dots$args, dots$keywords): ValueError: could not convert
import pandas as pd # Load pandas library
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No module
pd.to_numeric("abcd", errors = 'coerce') + 3
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not de
```

When we make our intent explicit (convert "abcd" to a numeric variable) we get an NA - a missing value - in R. In Python, we get a more descriptive error by default, but we can use the `pandas` library (which adds some statistical functionality) to get a similar result to the result we get in R.

There's still no easy way to figure out where "abcd" is on a number line, but our math will still have a result - `NA + 3` is `NA`.

6.6 Determining a Variable's Type

If you don't know what type a value is, both R and python have functions to help you with that.

R

If you are unsure what the type of a variable is, use the `typeof()` function to find out.

```r
w <- "a string"
x <- 3L
y <- 3.1415
z <- FALSE

typeof(w)
## [1] "character"
typeof(x)
## [1] "integer"
typeof(y)
## [1] "double"
typeof(z)
## [1] "logical"
```

Python

If you are unsure what the type of a variable is, use the `type()` function to find out.

```python
w = "a string"
x = 3
y = 3.1415
z = False

type(w)
## <class 'str'>
type(x)
## <class 'int'>
type(y)
## <class 'float'>
type(z)
## <class 'bool'>
```

Try It Out: Variables and Types

R

1. Create variables `string`, `integer`, `decimal`, and `logical`, with types that match the relevant vari-

able names.

```
string <-
integer <-
decimal <-
logical <-
```

2. Can you get rid of the error that occurs when this chunk is run?

```
logical + decimal
integer + decimal
string + integer
```

3. What happens when you add string to string? logical to logical?

Python

1. Create variables `string`, `integer`, `decimal`, and `logical`, with types that match the relevant variable names.

```
string =
integer =
decimal =
logical =
```

2. Can you get rid of the error that occurs when this chunk is run?

```
logical + decimal
integer + decimal
string + integer
```

3. What happens when you add string to string? logical to logical?

R Solution

```
string <- "hi, I'm a string"
integer <- 4L
decimal <- 5.412
logical <- TRUE

logical + decimal
## [1] 6.412
integer + decimal
## [1] 9.412
as.numeric(string) + integer
## [1] NA

"abcd" + "efgh"
## Error in "abcd" + "efgh": non-numeric argument to binary operator
TRUE + TRUE
## [1] 2
```

In R, adding a string to a string creates an error ("non-numeric argument to binary operator"). Adding a logical to a logical, e.g. TRUE + TRUE, results in 2, which is a numeric value.
To concatenate strings in R (like the default behavior in python), we would use the `paste0` function: `paste0("abcd", "efgh")`, which returns abcdefgh.

Python Solution

```
import pandas as pd
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
string = "hi, I'm a string"
integer = 4
decimal = 5.412
logical = True

logical + decimal
## 6.412
integer + decimal
## 9.411999999999999
pd.to_numeric(string, errors='coerce') + integer
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not
"abcd" + "efgh"
## 'abcdefgh'
True + True
## 2
```

In Python, when a string is added to another string, the two strings are **concatenated**. This differs from the result in R, which is a "non-numeric argument to binary operator" error.

6.7 References

7 Using Functions

In addition to variables, **functions** are extremely important in programming. Functions allow you to repeat a series of steps using different information and get the result. In a way, a function is to a variable as a verb is to a noun - functions are a concise way of performing an action.

7.1 Mathematical Operators

Let's first start with a special class of functions that you're probably familiar with from your math classes - mathematical operators.

Here are a few of the most important ones:

Operation	R symbol	Python symbol
Addition	`+`	`+`
Subtraction	`-`	`-`
Multiplication	`*`	`*`
Division	`/`	`/`
Integer Division	`%/%`	`//`
Modular Division	`%%`	`%`
Exponentiation	`^`	`**`

These operands are all for scalar operations (operations on a single number) - vectorized versions, such as matrix multiplication, are somewhat more complicated (and different between R and python).

Example: Integer and Modular Division

Integer division is the whole number answer to A/B, and modular division is the fractional remainder when A/B. Let's demonstrate with the problem 14/3, which evaluates to 4.6666667 when division is used, but has integer part 4 and remainder 2.

R

`14 %/% 3` in R would be 4, and `14 %% 3` in R would be 2.

```
14 %/% 3
## [1] 4
14 %% 3
## [1] 2
```

Python

```
14 // 3
## 4
14 % 3
## 2
```

7.2 Order of Operations

Both R and Python operate under the same mathematical rules of precedence that you learned in school. You may have learned the acronym PEMDAS, which stands for Parentheses, Exponents, Multiplication/Division, and Addition/Subtraction. That is, when examining a set of mathematical operations, we evaluate parentheses first, then exponents, and then we do multiplication/division, and finally, we add and subtract.

R

```
(1+1)^(5-2)
## [1] 8
1 + 2^3 * 4
## [1] 33
3*1^3
## [1] 3
```

Python

```
(1+1)**(5-2)
## 8
1 + 2**3*4
## 33
3*1**3
## 3
```

7.3 Simple String Operations

Python has some additional operators that work on strings. In R, you will have to use functions to perform these operations, as R does not have string operators.

Demo

Python

In Python, `+` will **concatenate** (stick together) two strings. Multiplying a string by an integer will repeat the string the specified number of times.

```
"first " + "second"
## 'first second'
"hello " * 3
## 'hello hello hello '
```

R

In R, to concatenate things, we need to use functions: `paste` or `paste0`:

```
paste("first", "second", sep = " ")
## [1] "first second"
paste("first", "second", collapse = " ")
## [1] "first second"
paste(c("first", "second"), sep = " ") # sep only works w/ 2 objects passed in
## [1] "first"  "second"
paste(c("first", "second"), collapse = " ") # collapse works on vectors
## [1] "first second"

paste(c("a", "b", "c", "d"),
      c("first", "second", "third", "fourth"),
      sep = "-", collapse = " ")
## [1] "a-first b-second c-third d-fourth"
# sep is used to collapse parameters, then collapse is used to collapse vectors

paste0(c("a", "b", "c"))
## [1] "a" "b" "c"
paste0("a", "b", "c") # equivalent to paste(..., sep = "")
## [1] "abc"
```

You don't need to understand the details of this code at this point in the class, but it is useful to know how to combine strings in both languages.

7.4 Using Functions

Functions are sets of instructions that take **arguments** and **return** values. Strictly speaking, mathematical operators (like those above) are a special type of functions – but we aren't going to get into that now.

We're also not going to talk about how to create our own functions just yet. Instead, I'm going to show you how to *use* functions.

Cheat Sheets!

It may be helpful at this point to print out the R reference card[a] and the Python reference card[b]. These cheat sheets contain useful functions for a variety of tasks in each language.

[a] From https://cran.r-project.org/doc/contrib/Short-refcard.pdf
[b] From http://sixthresearcher.com/wp-content/uploads/2016/12/Python3_reference_cheat_sheet.pdf

Methods are a special type of function that operate on a specific variable type. In Python, methods are applied using the syntax `variable.method_name()`. So, you can get the length of a string variable `my_string` using `my_string.length()`.

R has methods too, but they are invoked differently. In R, you would get the length of a string variable using `length(my_string)`.

Right now, it is not really necessary to know too much more about functions than this: you can invoke a function by passing in arguments, and the function will do a task and return the value.

Your Turn

7.4.0.1 Problem

Try out some of the functions mentioned on the R and Python cheatsheets.
Can you figure out how to define a list or vector of numbers? If so, can you use a function to calculate the maximum value?
Can you find the R functions that will allow you to repeat a string variable multiple times or concatenate two strings? Can you do this task in Python?

7.4.0.2 R Solution

```r
# Define a vector of numbers
x <- c(1, 2, 3, 4, 5)

# Calculate the maximum
max(x)
## [1] 5

# function to repeat a variable multiple times
rep("test", 3)
## [1] "test" "test" "test"
# Concatenate strings, using "ing... " as the separator
paste(rep("test", 3), collapse = "ing... ")
## [1] "testing... testing... test"
```

7.4.0.3 Python Solution

```python
# Define a list of numbers
x = [1, 2, 3, 4, 5]

# Calculate the maximum
max(x)

# Repeat a string multiple times
## 5
x = ("test", )*3 # String multiplication
                 # have to use a tuple () to get separate items
# Then use 'yyy'.join(x) to paste items of x together with yyy as separators
'ing... '.join(x)
## 'testing... testing... test'
```

7.5 Overpowerd Calculators

Now that you're familiar with how to use functions, if not how to define them, you are capable of using R or python as a very fancy calculator. Obviously, both languages can do many more

interesting things, which we'll get to, but let's see if we can make R and Python do some very basic stuff that hopefully isn't too foreign to you.

Example: Triangle Side Length

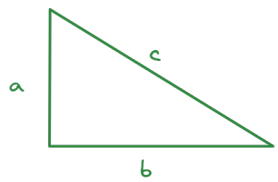


Figure 7.1: A right triangle with sides a, b, and hypotenuse c labeled.

Consider this triangle. I've measured the sides in an image editor and determined that $a = 212$ pixels, $b = 345$ pixels, and $c = 406$ pixels. I suspect, however, that my measurements aren't quite right - for one thing, I tried to measure in the center of the line, but it wasn't easy on the diagonal.
Let's assume that my measurements for a and b are accurate and calculate how far off my estimate was for side c.

7.5.0.1 R

```
# Define variables for the 3 sides of the triangle
a <- 212
b <- 345
c_meas <- 406
c_actual <- sqrt(a^2 + b^2)

# Calculate difference between measured and actual
# relative to actual
# and make it a percentage
pct_error <- (c_meas - c_actual)/c_actual * 100
pct_error
## [1] 0.2640307
```

7.5.0.2 Python

```
# To get the sqrt function, we have to import the math package
import math

# Define variables for the 3 sides of the triangle
a = 212
b = 345
c_meas = 406
c_actual = math.sqrt(a**2 + b**2)

# Calculate difference between measured and actual
# relative to actual
# and make it a percentage
pct_error = (c_meas - c_actual)/c_actual * 100
pct_error
## 0.264030681414134
```

Interesting, I wasn't as inaccurate as I thought!

Your Turn

Of course, if you remember trigonometry, we don't have to work with right triangles. Let's see if we can use trigonometric functions to do the same task with an oblique triangle.

7.5.0.3 Problem

Just in case you've forgotten your Trig, the Law of Cosines says that

$$c^2 = a^2 + b^2 - 2ab\cos(C),$$

where C is the angle between sides a and b.

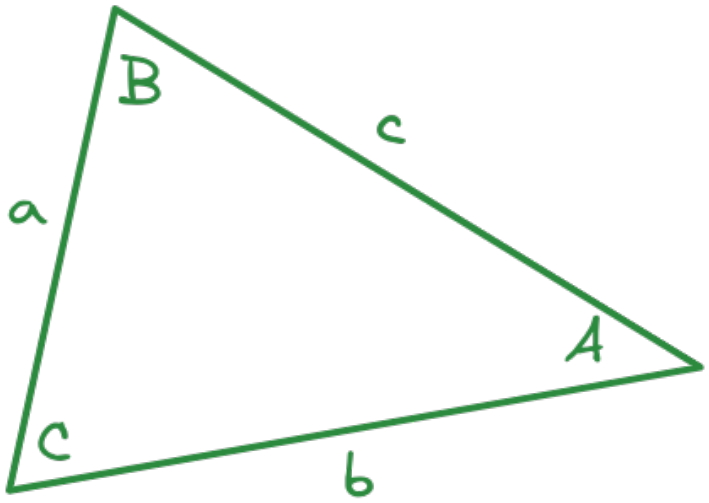


Figure 7.2: An oblique triangle with sides labeled a, b, and c, and angles labeled as A, B, C with capital letter opposite the lowercase side.

I measure side $a = 291$ pixels, side $b = 414$ pixels, and the angle between a and b to be 67.6°. What will I likely get for the length of side c in pixels?
Remember to check whether R and python compute trig functions using radians or degrees! As a reminder, π radians = 180°.

7.5.0.4 R solution

```
# Define variables for the 3 sides of the triangle
a <- 291
b <- 414
c_angle <- 67.6
c_actual <- sqrt(a^2 + b^2 - 2*a*b*cos(c_angle/180*pi))
c_actual
## [1] 405.2886
```

I measured the length of side c as 407 pixels.

7.5.0.5 Python solution

```
# To get the sqrt and cos functions, we have to import the math package
import math

# Define variables for the 3 sides of the triangle
a = 291
b = 414
c_angle = 67.6
c_actual = math.sqrt(a**2 + b**2 - 2*a*b*math.cos(c_angle/180*math.pi))
c_actual
## 405.28860699402117
```

I measured the length of side c as 407 pixels.

Congratulations, if you used a TI-83 in high school to do this sort of stuff, you're now just about as proficient with R and python as you were with that!

8 Vectors, Matrices, Arrays, and Control Structures

This chapter introduces some of the most important structures for storing and working with data: vectors, matrices, lists, and data frames.

8.1 Objectives

- Understand the differences between lists, vectors, data frames, matrices, and arrays in R and python
- Be able to use location-based indexing in R or python to pull out subsets of a complex data object
-

8.2 Data Structures Overview

In Chapter 6, we discussed 4 different data types: strings/characters, numeric/double/floats, integers, and logical/booleans. As you might imagine, things are about to get more complicated.

Data **structures** are more complex arrangements of information, but they are still (usually) created using the same data types we have previously discussed.

	Homogeneous	Heterogeneous
1D	vector	list
2D	matrix	data frame
N-D	array	

> **Warning**
>
> Those of you who have taken programming classes that were more computer science focused will realize that I am leaving out a lot of information about lower-level structures like pointers. I'm making a deliberate choice to gloss over most of those details in this chapter, because it's already hard enough to learn 2 languages worth of data structures at a time. In addition, R doesn't have pointers No Pointers in R, [11], so leaving out this material in python streamlines teaching both two languages, at the cost of overly simplifying some python concepts. If you want to read more about the Python concepts I'm leaving out, check out [12].

8.3 Lists

A **list** is a one-dimensional column of heterogeneous data - the things stored in a list can be of different types.

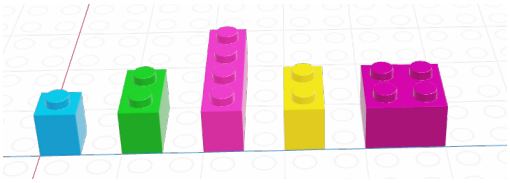


Figure 8.1: A lego list: the bricks are all different types and colors, but they are still part of the same data structure.

R

```
x <- list("a", 3, FALSE)
x
## [[1]]
## [1] "a"
##
## [[2]]
```

```
## [1] 3
##
## [[3]]
## [1] FALSE
```

Python

```
x = ["a", 3, False]
x
## ['a', 3, False]
```

The most important thing to know about lists, for the moment, is how to pull things out of the list. We call that process **indexing**.

8.3.1 Indexing

Every element in a list has an **index** (a location, indicated by an integer position)[1].

R concept

In R, we count from 1.

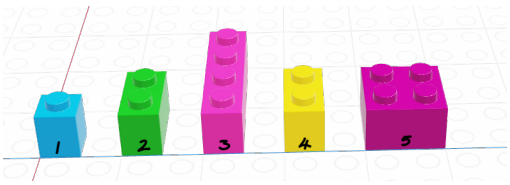


Figure 8.2: An R-indexed lego list, counting from 1 to 5

[1] Throughout this section (and other sections), lego pictures are rendered using https://www.mecabricks.com/en/workshop. It's a pretty nice tool for building stuff online!

R code

```r
x <- list("a", 3, FALSE)

x[1] # This returns a list
## [[1]]
## [1] "a"
x[1:2] # This returns multiple elements in the list
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 3

x[[1]] # This returns the item
## [1] "a"
x[[1:2]] # This doesn't work - you can only use [[]] with a single index
## Error in x[[1:2]]: subscript out of bounds
```

In R, list indexing with `[]` will return a list with the specified elements.

To actually retrieve the item in the list, use `[[]]`. The only downside to `[[]]` is that you can only access one thing at a time.

Python concept

In Python, we count from 0.

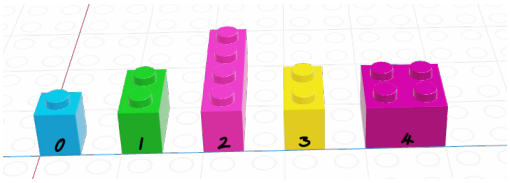


Figure 8.3: A python-indexed lego list, counting from 0 to 4

Python code

```
x = ["a", 3, False]

x[0]
## 'a'
x[1]
## 3
x[0:2]
## ['a', 3]
```

In Python, we can use single brackets to get an object or a list back out, but we have to know how **slices** work. Essentially, in Python, `0:2` indicates that we want objects 0 and 1, but want to stop at 2 (not including 2). If you use a slice, Python will return a list; if you use a single index, python just returns the value in that location in the list.

We'll talk more about indexing as it relates to vectors, but indexing is a general concept that applies to just about any multi-value object.

8.4 Vectors

A **vector** is a one-dimensional column of homogeneous data. **Homogeneous** means that every element in a vector has the same data type.

We can have vectors of any data type and length we want:

8.4.1 Indexing by Location

Each element in a vector has an **index** - an integer telling you what the item's position within the vector is. I'm going to demonstrate indices with the string vector

R	Python
1-indexed language	0-indexed language

R	Python
Count elements as 1, 2, 3, 4, …, N	Count elements as 0, 1, 2, 3, , …, N-1
	

R

In R, we create vectors with the `c()` function, which stands for "concatenate" - basically, we stick a bunch of objects into a row.

```
digits_pi <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)

# Access individual entries
digits_pi[1]
## [1] 3
digits_pi[2]
## [1] 1
digits_pi[3]
## [1] 4

# R is 1-indexed - a list of 11 things goes from 1 to 11
digits_pi[0]
## numeric(0)
digits_pi[11]
## [1] 5

# Print out the vector
digits_pi
##  [1] 3 1 4 1 5 9 2 6 5 3 5
```

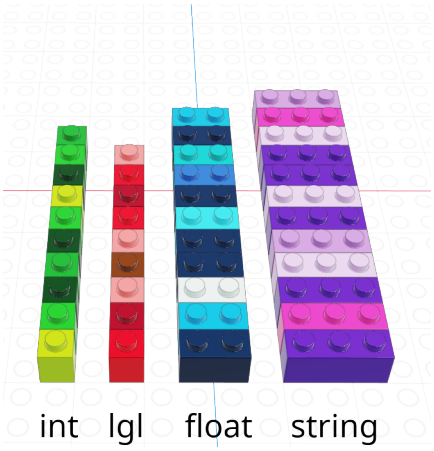


Figure 8.4: vectors of different data types

Python Vectors

In python, we create vectors using the `array` function in the numpy module. To add a python module, we use the syntax `import <name> as <nickname>`. Many modules have conventional (and very short) nicknames - for `numpy`, we will use `np` as the nickname. Any functions we reference in the `numpy` module will then be called using `np.fun_name()` so that python knows where to find them.[2]

```
import numpy as np
digits_list = [3,1,4,1,5,9,2,6,5,3,5]
digits_pi = np.array(digits_list)

# Access individual entries
digits_pi[0]
## 3
digits_pi[1]
## 1
digits_pi[2]
```

[2] A similar system exists in R libraries, but R doesn't handle multiple libraries having the same function names well, which leads to all sorts of confusion. At least python is explicit about it.

```
# Python is 0 indexed - a list of 11 things goes from 0 to 10
## 4
digits_pi[0]
## 3
digits_pi[11]

# multiplication works on the whole vector at once
## Error in py_call_impl(callable, dots$args, dots$keywords): IndexError: index 11 is out of
digits_pi * 2

# Print out the vector
## array([ 6,  2,  8,  2, 10, 18,  4, 12, 10,  6, 10])
print(digits_pi)
## [3 1 4 1 5 9 2 6 5 3 5]
```

8.4.1.1 Python Series (Pandas)

Python has multiple things that look like vectors, including the `pandas` library's Series structure. A **Series** is a one-dimensional array-like object containing a sequence of values and an associated array of labels (called its index).

```
import pandas as pd
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No module
digits_pi = pd.Series([3,1,4,1,5,9,2,6,5,3,5])

# Access individual entries
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not de
digits_pi[0]
## 3
digits_pi[1]
## 1
digits_pi[2]

# Python is 0 indexed - a list of 11 things goes from 0 to 10
## 4
digits_pi[0]
## 3
digits_pi[11]
```

```
# logical indexing works here too
## Error in py_call_impl(callable, dots$args, dots$keywords): IndexError: index 11 is out of
digits_pi[digits_pi > 3]
# simple multiplication works in a vectorized manner
# that is, the whole vector is multiplied at once
## array([4, 5, 9, 6, 5, 5])
digits_pi * 2

# Print out the series
## array([ 6,  2,  8,  2, 10, 18,  4, 12, 10,  6, 10])
print(digits_pi)
## [3 1 4 1 5 9 2 6 5 3 5]
```

The Series object has a list of labels in the first printed column, and a list of values in the second. If we want, we can specify the labels manually to use as e.g. plot labels later:

```
import pandas as pd
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No module
weekdays = pd.Series(['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Sat

# access individual objs
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not de
weekdays[0]
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
weekdays[1]
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
weekdays['S']
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
weekdays['Sat']

# access the index
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
weekdays.index
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
weekdays.index[6] = 'Z' # you can't assign things to the index to change it
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
weekdays
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'weekdays' is
```

We can pull out items in a vector by indexing, but we can also replace specific things as well:

R

```
favorite_cats <- c("Grumpy", "Garfield", "Jorts", "Jean")

favorite_cats
## [1] "Grumpy"   "Garfield" "Jorts"    "Jean"

favorite_cats[2] <- "Nyan Cat"

favorite_cats
## [1] "Grumpy"   "Nyan Cat" "Jorts"    "Jean"
```

Python

```
favorite_cats = ["Grumpy", "Garfield", "Jorts", "Jean"]

favorite_cats
## ['Grumpy', 'Garfield', 'Jorts', 'Jean']
favorite_cats[1] = "Nyan Cat"

favorite_cats
## ['Grumpy', 'Nyan Cat', 'Jorts', 'Jean']
```

If you're curious about any of these cats, see the footnotes[3].

8.4.2 Indexing with Logical Vectors

As you might imagine, we can create vectors of all sorts of different data types. One particularly useful trick is to create a **logical vector** that goes along with a vector of another type to use as a **logical index**.

[3] Grumpy cat, Garfield, Nyan cat. Jorts and Jean: The initial post and the update (both are worth a read because the story is hilarious). The cats also have a Twitter account where they promote workers rights.

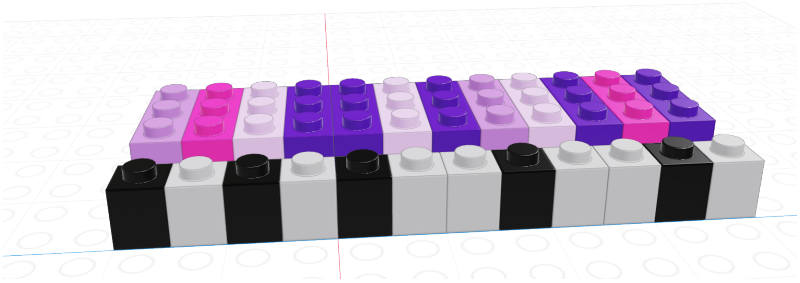
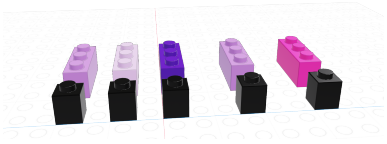
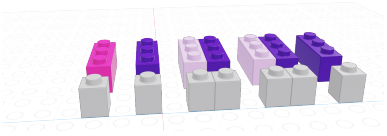


Figure 8.5: lego vectors - a pink/purple hued set of 1x3 bricks representing the data and a corresponding set of 1x1 grey and black bricks representing the logical index vector of the same length

If we let the black lego represent "True" and the grey lego represent "False", we can use the logical vector to pull out all values in the main vector.

Black = True, Grey = False	Grey = True, Black = False
	

Note that for logical indexing to work properly, the logical index must be the same length as the vector we're indexing. This constraint will return when we talk about data frames, but for now just keep in mind that logical indexing doesn't make sense when this constraint isn't true.

Indexing with logical vectors in R

```r
# Define a character vector
weekdays <- c("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday")
weekend <- c("Sunday", "Saturday")

# Create logical vectors
relax_days <- c(1, 0, 0, 0, 0, 0, 1) # doing this the manual way
relax_days <- weekdays %in% weekend # This creates a logical vector
                                    # with less manual construction
relax_days
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE

school_days <- !relax_days # FALSE if weekend, TRUE if not
school_days
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

# Using logical vectors to index the character vector
weekdays[school_days] # print out all school days
## [1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"
```

Indexing with logical vectors in python

```python
import numpy as np;

animals = np.array(["Cat", "Dog", "Snake", "Lizard", "Tarantula", "Hamster", "Gerbil", "Otte

# Define a logical vector
good_pets = np.array([True, True, False, False, False, True, True, False])
bad_pets = np.invert(good_pets) # Invert the logical vector
                                # so True -> False and False -> True

animals[good_pets]
## array(['Cat', 'Dog', 'Hamster', 'Gerbil'], dtype='<U9')
animals[bad_pets]
## array(['Snake', 'Lizard', 'Tarantula', 'Otter'], dtype='<U9')
animals[~good_pets] # equivalent to using bad_pets
## array(['Snake', 'Lizard', 'Tarantula', 'Otter'], dtype='<U9')
```

8.4.3 Reviewing Types

As vectors are a collection of things of a single type, what happens if we try to make a vector with differently-typed things?

R

```
c(2L, FALSE, 3.1415, "animal") # all converted to strings
## [1] "2"      "FALSE"  "3.1415" "animal"

c(2L, FALSE, 3.1415) # converted to numerics
## [1] 2.0000 0.0000 3.1415

c(2L, FALSE) # converted to integers
## [1] 2 0
```

Python

```
import numpy as np

np.array([2, False, 3.1415, "animal"]) # all converted to strings
## array(['2', 'False', '3.1415', 'animal'], dtype='<U32')
np.array([2, False, 3.1415]) # converted to floats
## array([2.    , 0.    , 3.1415])
np.array([2, False]) # converted to integers
## array([2, 0])
```

As a reminder, this is an example of **implicit** type conversion - R and python decide what type to use for you, going with the type that doesn't lose data but takes up as little space as possible.

Try it Out!

Problem

Create a vector of the integers from one to 30. Use logical indexing to pick out only the numbers which are multiples of

3.

R Solution

```
x <- 1:30
x [ x %% 3 == 0]
##  [1]  3  6  9 12 15 18 21 24 27 30
```

Python Solution

```
import numpy as np
x = np.array(range(1, 31)) # because python is 0 indexed
x[ x % 3 == 0]
## array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30])
```

8.4.3.1 Challenge

Extra challenge: Pick out numbers which are multiples of 2 or 3, but not multiples of 6!

General Solution

This operation is **xor**, a.k.a. exclusive or. That is, X or Y, but not X AND Y.

We can write xor as `(X OR Y) & !(X AND Y)` – or we can use a predefined function: `xor()` in R, `^` in python.

R Solution

```
x <- 1:30

x2 <- x %% 2 == 0 # multiples of 2
x3 <- x %% 3 == 0 # multiples of 3
x2xor3 <- xor(x2, x3)
```

```r
x2xor3_2 <- (x2 | x3) & !(x2 & x3)
x[x2xor3]
##  [1]  2  3  4  8  9 10 14 15 16 20 21 22 26 27 28
x[x2xor3_2]
##  [1]  2  3  4  8  9 10 14 15 16 20 21 22 26 27 28
```

Python Solution

```python
import numpy as np
x = np.array(range(1, 31))

x2 = x % 2 == 0 # multiples of 2
x3 = x % 3 == 0 # multiples of 3
x2xor3 = x2 ^ x3

x[x2xor3]
## array([ 2,  3,  4,  8,  9, 10, 14, 15, 16, 20, 21, 22, 26, 27, 28])
```

8.5 Matrices

A **matrix** is the next step after a vector - it's a set of values arranged in a two-dimensional, rectangular format.

Matrix (Lego)

R

```r
# Minimal matrix in R: take a vector,
# tell R how many rows you want
matrix(1:12, nrow = 3)
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

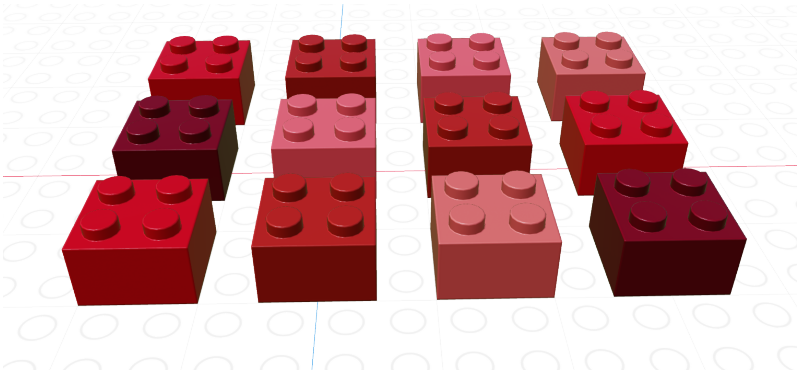


Figure 8.6: lego depiction of a 3-row, 4-column matrix of 2x2 red-colored blocks

```
matrix(1:12, ncol = 3) # or columns
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12

# by default, R will fill in column-by-column
# the byrow parameter tells R to go row-by-row
matrix(1:12, nrow = 3, byrow = T)
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12

# We can also easily create square matrices
# with a specific diagonal (this is useful for modeling)
diag(rep(1, times = 4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

Python

In python, matrices are just a special case of a class called `ndarray` - n-dimensional arrays.

```
import numpy as np
# Minimal ndarray in python by typing in the values in a structured format
np.array([[0,  1,  2],
          [3,  4,  5],
          [6,  7,  8],
          [9, 10, 11]])
# This syntax creates a list of the rows we want in our matrix

# Matrix in python using a data vector and size parameters
## array([[ 0,  1,  2],
##        [ 3,  4,  5],
##        [ 6,  7,  8],
##        [ 9, 10, 11]])
np.reshape(range(0,12), (3,4))
## array([[ 0,  1,  2,  3],
##        [ 4,  5,  6,  7],
##        [ 8,  9, 10, 11]])
np.reshape(range(0,12), (4,3))
## array([[ 0,  1,  2],
##        [ 3,  4,  5],
##        [ 6,  7,  8],
##        [ 9, 10, 11]])
np.reshape(range(0,12), (3,4), order = 'F')
## array([[ 0,  3,  6,  9],
##        [ 1,  4,  7, 10],
##        [ 2,  5,  8, 11]])
```

In python, we create 2-dimensional arrays (aka matrices) either by creating a list of rows to join together or by reshaping a 1-dimensional array. The trick with reshaping the 1-dimensional array is the order argument: 'F' stands for "Fortran-like" and 'C' stands for "C-like"... so to go by column, you use 'F' and to go by row, you use 'C'. Totally intuitive, right?

Most of the problems we're going to work on will not require much in the way of matrix or array operations. For now, you

need the following:

- Know that matrices exist and what they are (2-dimensional arrays of numbers)
- Understand how they are indexed (because it is extremely similar to data frames that we'll work with in the next chapter)
- Be aware that there are lots of functions that depend on matrix operations at their core (including linear regression)

For more on matrix operations and matrix calculations, see Chapter 14.

8.5.1 Indexing in Matrices

Both R and python use [row, column] to index matrices. To extract the bottom-left element of a 3x4 matrix in R, we would use [3,1] to get to the third row and first column entry; in python, we would use [2,0] (remember that Python is 0-indexed).

As with vectors, you can replace elements in a matrix using assignment.

R

```
my_mat <- matrix(1:12, nrow = 3, byrow = T)

my_mat[3,1] <- 500

my_mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]  500   10   11   12
```

Python

Remember that zero-indexing!

```
import numpy as np

my_mat = np.reshape(range(1, 13), (3,4))

my_mat[2,0] = 500

my_mat
## array([[  1,   2,   3,   4],
##        [  5,   6,   7,   8],
##        [500,  10,  11,  12]])
```

8.5.2 Matrix Operations

There are a number of matrix operations that we need to know for basic programming purposes:

- scalar multiplication

$$c * \mathbf{X} = c * \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} = \begin{bmatrix} c * x_{1,1} & c * x_{1,2} \\ c * x_{2,1} & c * x_{2,2} \end{bmatrix}$$

- transpose - flip the matrix across the left top -> right bottom diagonal.

$$t(\mathbf{X}) = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix}^T = \begin{bmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{bmatrix}$$

- matrix multiplication (dot product) - If you haven't had this in Linear Algebra, here's a preview. See [13] for a better explanation

$$\mathbf{X} * \mathbf{Y} = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix} * \begin{bmatrix} y_{1,1} \\ y_{2,1} \end{bmatrix} = \begin{bmatrix} x_{1,1} * y_{1,1} + x_{1,2} * y_{2,1} \\ x_{2,1} * y_{1,1} + x_{2,2} * y_{2,1} \end{bmatrix}$$

  Note that matrix multiplication depends on having matrices of compatible dimensions. If you have two matrices of dimension $(a \times b)$ and $(c \times d)$, then b must be equal to c for the multiplication to work, and your result will be $(a \times d)$.

R

```r
x <- matrix(c(1, 2, 3, 4), nrow = 2, byrow = T)
y <- matrix(c(5, 6), nrow = 2)

# Scalar multiplication
x * 3
##      [,1] [,2]
## [1,]    3    6
## [2,]    9   12
3 * x
##      [,1] [,2]
## [1,]    3    6
## [2,]    9   12

# Transpose
t(x)
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
t(y)
##      [,1] [,2]
## [1,]    5    6

# matrix multiplication (dot product)
x %*% y
##      [,1]
## [1,]   17
## [2,]   39
```

Python

```python
import numpy as np
x = np.array([[1,2],[3,4]])
y = np.array([[5],[6]])

# scalar multiplication
x*3
```

```
## array([[ 3,  6],
##        [ 9, 12]])
3*x

# transpose
## array([[ 3,  6],
##        [ 9, 12]])
x.T # shorthand
## array([[1, 3],
##        [2, 4]])
x.transpose() # Long form

# Matrix multiplication (dot product)
## array([[1, 3],
##        [2, 4]])
np.dot(x, y)
## array([[17],
##        [39]])
```

8.6 Arrays

Arrays are a generalized n-dimensional version of a vector: all elements have the same type, and they are indexed using square brackets in both R and python: `[dim1, dim2, dim3, ...]`

I don't think you will need to create 3+ dimensional arrays in this class, but if you want to try it out, here is some code.

R

```
array(1:8, dim = c(2,2,2))
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
```

```
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

Note that displaying this requires 2 slices, since it's hard to display 3D information in a 2D terminal arrangement.

Python

```
import numpy as np

np.array([[[1,2],[3,4]],[[5,6], [7,8]]])
## array([[[1, 2],
##         [3, 4]],
##
##        [[5, 6],
##         [7, 8]]])
```

8.7 Data Frames

A data frame is a special type of list - one in which each element in the list is a vector of the same length. If you put these vectors side-by-side, you get a table of data that looks like a spreadsheet. In Python, a DataFrame is a dict of Series.

The lego version of a data frame looks like this:

R

When you examine the structure of a data frame, as shown below, you get each column shown in a row, with its type and the first few values in the column. The `head(n)` command shows the first n rows of a data frame (enough to see what's there, not enough to overflow your screen).

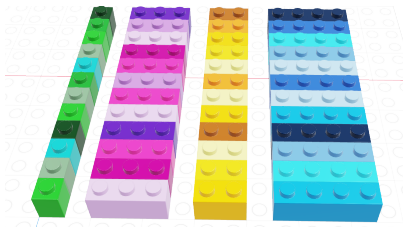


Figure 8.7: A data frame with 4 columns. A data frame is essentially a list where all of the components are vectors or lists, and are constrained to have the same length.

```
data(mtcars) # Load the data -- included in base R
head(mtcars) # Look at the first 6 rows
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
str(mtcars) # Examine the structure of the object
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

You can change column values or add new columns easily using assignment. The `summary()` function can be used on specific columns to perform summary operations (a 5-number summary useful for making e.g. boxplots is provided by default).

```
mtcars$gpm <- 1/mtcars$mpg # gpm is sometimes used to assess efficiency

summary(mtcars$gpm)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.02950 0.04386 0.05208 0.05423 0.06483 0.09615
summary(mtcars$mpg)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.40   15.43   19.20   20.09   22.80   33.90
```

Often, it is useful to know the dimensions of a data frame. The number of rows can be obtained by using `nrow(df)` and similarly, the columns can be obtained using `ncol(df)` (or, get both with `dim()`). There is also an easy way to get a summary of each column in the data frame, using `summary()`.

```
summary(mtcars)
##       mpg             cyl             disp             hp
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
##       drat             wt             qsec             vs
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
##        am              gear            carb             gpm
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000   Min.   :0.02950
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000   1st Qu.:0.04386
##  Median :0.0000   Median :4.000   Median :2.000   Median :0.05208
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812   Mean   :0.05423
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:0.06483
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000   Max.   :0.09615
dim(mtcars)
## [1] 32 12
nrow(mtcars)
```

```
## [1] 32
ncol(mtcars)
## [1] 12
```

Missing variables in an R data frame are indicated with `NA`.

Python

When you examine the structure of a data frame, as shown below, you get each column shown in a row, with its type and the first few values in the column. The `df.head(n)` command shows the first n rows of a data frame (enough to see what's there, not enough to overflow your screen).

```
mtcars = pd.read_csv("https://vincentarelbundock.github.io/Rdatasets/csv/datasets/mtcars.csv
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not de
mtcars.head(5)
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
mtcars.info()
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
```

You can change column values or add new columns easily using assignment. It's also easy to access specific columns to perform summary operations. You can access a column named xyz using `df.xyz` or using `df["xyz"]`. To create a new column, you must use `df["xyz"]`.

```
mtcars["gpm"] = 1/mtcars.mpg # gpm is sometimes used to assess efficiency
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
mtcars.gpm.describe()
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
mtcars.mpg.describe()
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
```

Often, it is useful to know the dimensions of a data frame. The dimensions of a data frame (rows x columns) can be accessed using `df.shape`. There is also an easy way to get a summary of each column in the data frame, using `df.describe()`.

```
mtcars.describe()
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
mtcars.shape
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'mtcars' is no
```

Missing variables in a pandas data frame are indicated with `nan` or `NULL`.

Try it out: Data Frames

Setup

The dataset `state.x77` contains information on US state statistics in the 1970s. By default, it is a matrix, but we can easily convert it to a data frame, as shown below.

```
data(state)
state_facts <- data.frame(state.x77)
state_facts <- cbind(state = row.names(state_facts), state_facts, stringsAsFactors = F)
# State names were stored as row labels
# Store them in a variable instead, and add it to the data frame

row.names(state_facts) <- NULL # get rid of row names

head(state_facts)
##        state Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Area
## 1    Alabama       3615   3624        2.1    69.05   15.1    41.3    20  50708
## 2     Alaska        365   6315        1.5    69.31   11.3    66.7   152 566432
## 3    Arizona       2212   4530        1.8    70.55    7.8    58.1    15 113417
## 4   Arkansas       2110   3378        1.9    70.66   10.1    39.9    65  51945
## 5 California      21198   5114        1.1    71.71   10.3    62.6    20 156361
## 6   Colorado       2541   4884        0.7    72.06    6.8    63.9   166 103766

# Write data out so that we can read it in using Python
write.csv(state_facts, file = "data/state_facts.csv", row.names = F)
## Error in file(file, ifelse(append, "a", "w")): cannot open the connection
```

We can write out the built in R data and read it in using `pd.read_csv`, which creates a DataFrame in pandas.

```
import pandas as pd
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
state_facts = pd.read_csv("https://raw.githubusercontent.com/srvanderplas/unl-stat850/main
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not
```

Problem

1. How many rows and columns does it have? Can you find different ways to get that information?
2. The `Illiteracy` column contains the percent of the population of each state that is illiterate. Calculate the number of people in each state who are illiterate, and store that in a new column called `TotalNumIlliterate`. Note: `Population` contains the population in thousands.
3. Calculate the average population density of each state (population per square mile) and store it in a new column `PopDensity`. Using the R reference card, can you find functions that you can combine to get the state with the minimum population density?

R Solution

```
# 3 ways to get rows and columns
str(state_facts)
## 'data.frame':    50 obs. of  9 variables:
##  $ state     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  $ Population: num  3615 365 2212 2110 21198 ...
##  $ Income    : num  3624 6315 4530 3378 5114 ...
##  $ Illiteracy: num  2.1 1.5 1.8 1.9 1.1 0.7 1.1 0.9 1.3 2 ...
##  $ Life.Exp  : num  69 69.3 70.5 70.7 71.7 ...
##  $ Murder    : num  15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 10.7 13.9 ...
##  $ HS.Grad   : num  41.3 66.7 58.1 39.9 62.6 63.9 56 54.6 52.6 40.6 ...
##  $ Frost     : num  20 152 15 65 20 166 139 103 11 60 ...
##  $ Area      : num  50708 566432 113417 51945 156361 ...
dim(state_facts)
## [1] 50  9
nrow(state_facts)
## [1] 50
ncol(state_facts)
## [1] 9

# Illiteracy
state_facts$TotalNumIlliterate <- state_facts$Population * 1e3 * (state_facts$Illiteracy/1

# Population Density
state_facts$PopDensity <- state_facts$Population * 1e3/state_facts$Area
# in people per square mile

# minimum population
state_facts$state[which.min(state_facts$PopDensity)]
## [1] "Alaska"
```

Python Solution

```
# Ways to get rows and columns
state_facts.shape
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
state_facts.index.size # rows
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
state_facts.columns.size # columns
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
state_facts.info() # columns + rows + missing counts + data types

# Illiteracy
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
state_facts["TotalNumIlliterate"] = state_facts["Population"] * 1e3 * state_facts["Illiter

# Population Density
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
state_facts["PopDensity"] = state_facts["Population"] * 1e3/state_facts["Area"]
# in people per square mile

# minimum population
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
min_dens = state_facts["PopDensity"].min()
# Get location of minimum population
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
loc_min_dens = state_facts.PopDensity.isin([min_dens])
# Pull out matching state
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
state_facts.state[loc_min_dens]
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'state_facts
```

8.7.1 Creating Data Frames

It is also possible to create data frames from scratch by building them out of simpler components, such as vectors or dicts of lists. This tends to be useful for small data sets, but it is more common to read data in from e.g. CSV files, which I've used several times already but haven't yet shown you how to do (see Chapter 15 for the full how-to).

R

```
math_and_lsd <- data.frame(
  lsd_conc = c(1.17, 2.97, 3.26, 4.69, 5.83, 6.00, 6.41),
  test_score = c(78.93, 58.20, 67.47, 37.47, 45.65, 32.92, 29.97))
math_and_lsd
##   lsd_conc test_score
## 1     1.17      78.93
## 2     2.97      58.20
## 3     3.26      67.47
## 4     4.69      37.47
## 5     5.83      45.65
## 6     6.00      32.92
## 7     6.41      29.97

# add a column - character vector
math_and_lsd$subjective <- c("finally coming back", "getting better", "it's totally better",

math_and_lsd
##   lsd_conc test_score                              subjective
## 1     1.17      78.93                    finally coming back
## 2     2.97      58.20                         getting better
## 3     3.26      67.47                    it's totally better
## 4     4.69      37.47                    really tripping out
## 5     5.83      45.65                            is it over?
## 6     6.00      32.92                              whoa, man
## 7     6.41      29.97 I can taste color, but I can't do math
```

Python

```
math_and_lsd = pd.DataFrame({
  "lsd_conc": [1.17, 2.97, 3.26, 4.69, 5.83, 6.00, 6.41],
  "test_score": [78.93, 58.20, 67.47, 37.47, 45.65, 32.92, 29.97]})
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not de
math_and_lsd

# add a column - character vector
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'math_and_lsd'
```

```
math_and_lsd["subjective"] = ["finally coming back", "getting better", "it's totally better"
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'math_and_lsd'
math_and_lsd
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'math_and_lsd'
```

8.8 References

9 Control Structures

Control structures are statements in a program that determine when code is evaluated (and how many times it might be evaluated). There are two main types of control structures: if-statements and loops.

Before we start on the types of control structures, let's get in the right mindset. We're all used to "if-then" logic, and use it in everyday conversation, but computers require another level of specificity when you're trying to provide instructions.

Check out this video of the classic "make a peanut butter sandwich instructions challenge":

Here's another example:

The key takeaways from these bits of media are that you should read this section with a focus on exact precision - state *exactly* what you mean, and the computer will do what you say. If you instead expect the computer to get what you mean, you're going to have a bad time.

9.1 Conditional Statements

Conditional statements determine if code is evaluated.

They look like this:

```
if (condition)
  then
    (thing to do)
  else
    (other thing to do)
```

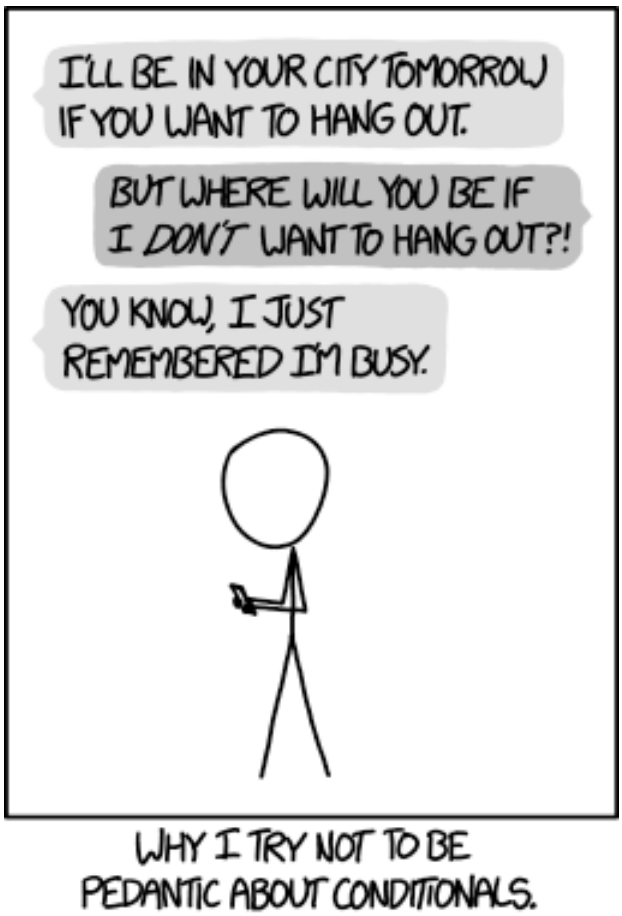


Figure 9.1: 'If you're done being pedantic, we should get dinner.' 'You did it again!' 'No, I didn't.' Image from Randal Munroe, xkcd.com, available under a CC-By 2.5 license.

The else (other thing to do) part may be omitted.

When this statement is read by the computer, the computer checks to see if condition is true or false. If the condition is true, then (thing to do) is also run. If the condition is false, then (other thing to do) is run instead.

Let's try this out:

R

```
x <- 3
y <- 1

if (x > 2) {
  y <- 8
} else {
  y <- 4
}

print(paste("x =", x, "; y =", y))
## [1] "x = 3 ; y = 8"
```

In R, the logical condition after `if` must be in parentheses. It is common to then enclose the statement to be run if the condition is true in `{}` so that it is clear what code matches the if statement. You can technically put the condition on the line after the `if (x > 2)` line, and everything will still work, but then it gets hard to figure out what to do with the else statement - it technically would also go on the same line, and that gets hard to read.

```
x <- 3
y <- 1

if (x > 2) y <- 8 else y <- 4

print(paste("x =", x, "; y =", y))
## [1] "x = 3 ; y = 8"
```

So while the 2nd version of the code technically works, the first version with the brackets is much easier to read and understand. Please try to emulate the first version!

Python

```
x = 3
y = 1

if x > 2:
  y = 8
else:
  y = 4

print("x =", x, "; y =", y)
## x = 3 ; y = 8
```

In python, all code grouping is accomplished with spaces instead of with brackets. So in python, we write our if statement as `if x > 2:` with the colon indicating that what follows is the code to evaluate. The next line is indented with 2 spaces to show that the code on those lines belongs to that if statement. Then, we use the else: statement to provide an alternative set of code to run if the logical condition in the if statement is false. Again, we indent the code under the else statement to show where it "belongs".

> **Warning**
>
> Python will throw errors if you mess up the spacing. This is one thing that is very annoying about Python… but it's a consequence of trying to make the code more readable.

9.1.1 Representing Conditional Statements as Diagrams

A common way to represent conditional logic is to draw a flow chart diagram.

In a flow chart, conditional statements are represented as diamonds, and other code is represented as a rectangle. Yes/no or True/False branches are labeled. Typically, after a conditional statement, the program flow returns to a single point.

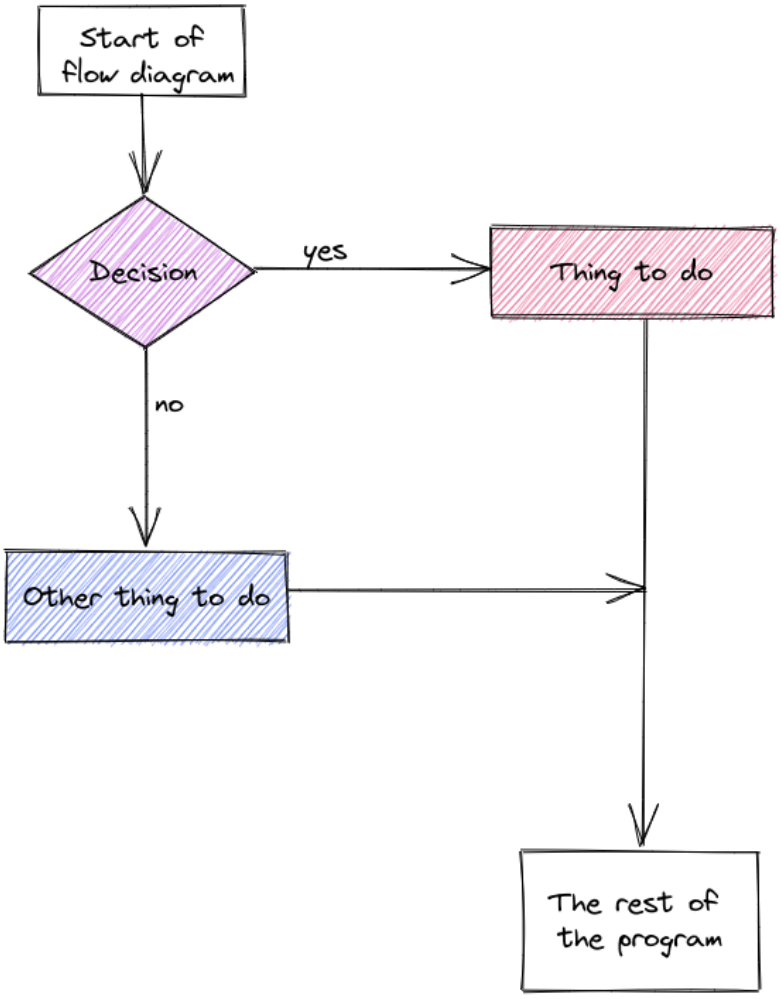


Figure 9.2: Program flow diagram outline of a simple if/else statement

9.1.2 Chaining Conditional Statements: Else-If

In many cases, it can be helpful to have a long chain of conditional statements describing a sequence of alternative statements.

Example - Conditional Evaluation

Suppose I want to determine what categorical age bracket someone falls into based on their numerical age. All of the bins are mutually exclusive - you can't be in the 25-40 bracket and the 41-55 bracket.

Program Flow Map

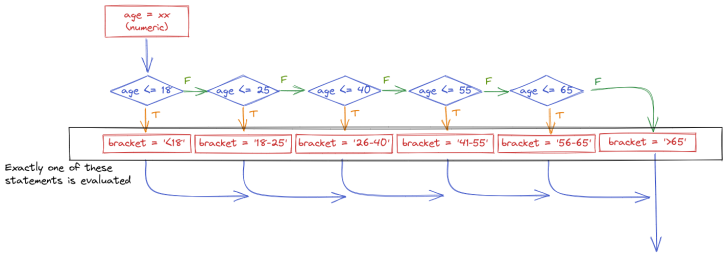


Figure 9.3: Program flow map for a series of mutually exclusive categories. If our goal is to take a numeric age variable and create a categorical set of age brackets, such as <18, 18-25, 26-40, 41-55, 56-65, and >65, we can do this with a series of if-else statements chained together. Only one of the bracket assignments is evaluated, so it is important to place the most restrictive condition first.

The important thing to realize when examining this program flow map is that if `age <= 18` is true, then **none of the other conditional statements even get evaluated**. That is, once a statement is true, none of the other statements matter. Because of this, it is important to place the most restrictive statement first.

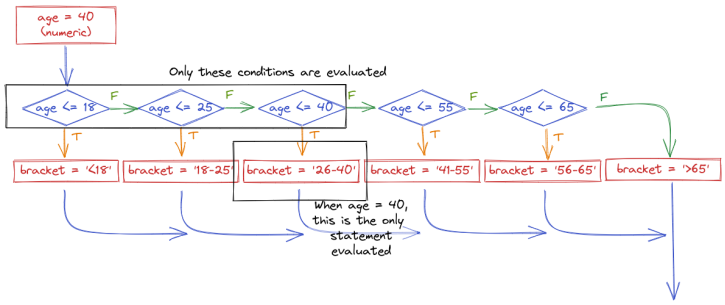


Figure 9.4: Program flow map for a series of mutually exclusive categories, emphasizing that only some statements are evaluated. When age = 40, only (age <= 18), (age <= 25), and (age <= 40) are evaluated conditionally. Of the assignment statements, only bracket = '26-40' is evaluated when age = 40.

If for some reason you wrote your conditional statements in the wrong order, the wrong label would get assigned:

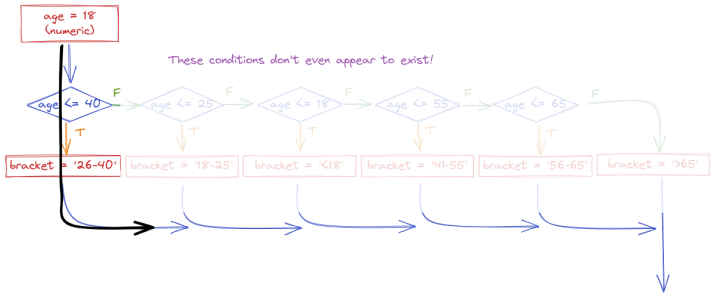


Figure 9.5: Program flow map for a series of mutually exclusive categories, with category labels in the wrong order - <40 is evaluated first, and so <= 25 and <= 18 will never be evaluated and the wrong label will be assigned for anything in those categories.

In code, we would write this statement using `else-if` (or `elif`) statements.

R

```
age <- 40 # change this as you will to see how the code works

if (age < 18) {
  bracket <- "<18"
} else if (age <= 25) {
  bracket <- "18-25"
} else if (age <= 40) {
  bracket <- "26-40"
} else if (age <= 55) {
  bracket <- "41-55"
} else if (age <= 65) {
  bracket <- "56-65"
} else {
  bracket <- ">65"
}

bracket
## [1] "26-40"
```

Python

Python uses `elif` as a shorthand for `else if` statements. As always, indentation/white space in python matters. If you put an extra blank line between two elif statements, then the interpreter will complain. If you don't indent properly, the interpreter will complain.

```r
age = 40 # change this to see how the code works

if age < 18:
  bracket = "<18"
elif age <= 25:
  bracket = "18-25"
elif age <= 40:
  bracket = "26-40"
elif age <= 55:
  bracket = "41-55"
elif age <= 65:
  bracket = "56-65"
else:
  bracket = ">65"

bracket
## '26-40'
```

Try it out - Chained If/Else Statements

Problem

The US Tax code has brackets, such that the first $10,275 of your income is taxed at 10%, anything between $10,275 and $41,775 is taxed at 12%, and so on.
Here is the table of tax brackets for single filers in 2022:

rate	Income
10%	$0 to $10,275
12%	$10,275 to $41,775
22%	$41,775 to $89,075
24%	$89,075 to $170,050
32%	$170,050 to $215,950
35%	$215,950 to $539,900
37%	$539,900 or more

Note: For the purposes of this problem, we're ignoring the personal exemption and the standard deduction, so we're already simplifying the tax code.
Write a set of if statements that assess someone's income and determine what their overall tax rate is.
Hint: You may want to keep track of how much of the income has already been taxed in a variable and what the total tax accumulation is in another variable.

9.1.2.1 Flow Map

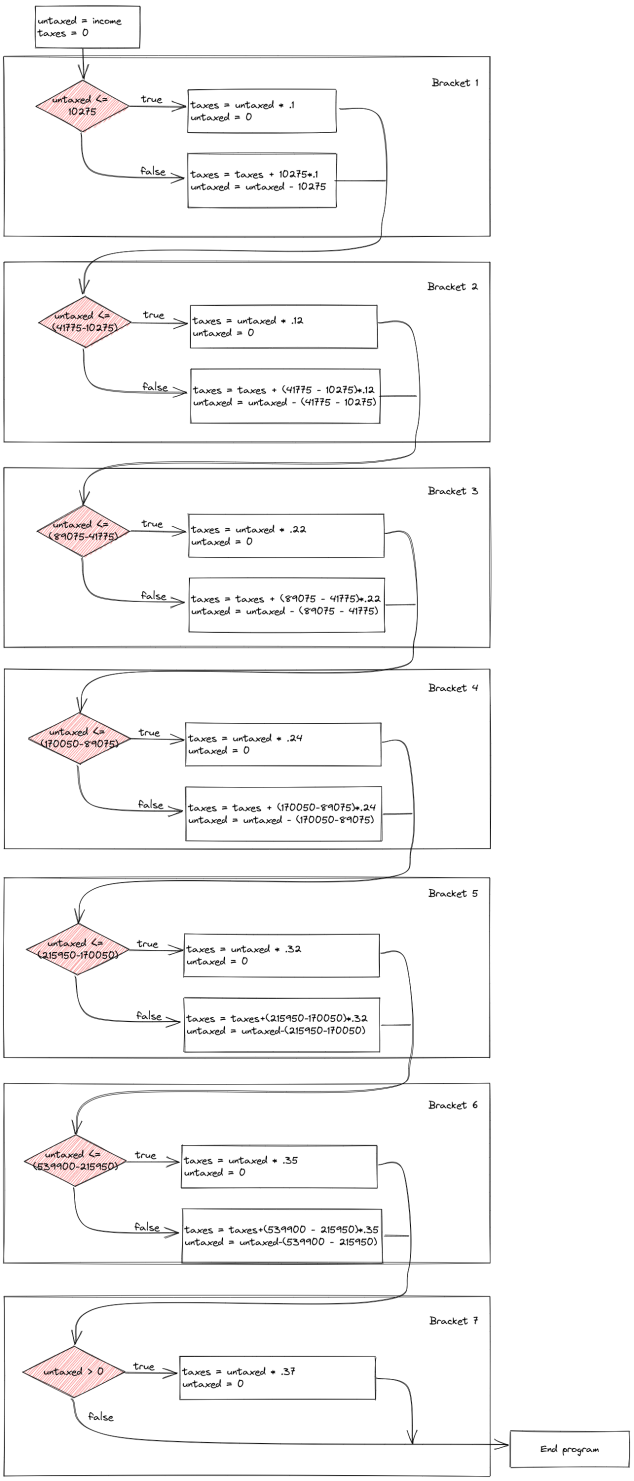


Figure 9.6: The control flow diagram for the tax brackets

Control flow diagrams can be extremely helpful when figuring out how programs work (and where gaps in your logic are when you're debugging). It can be very helpful to map out your program flow as you're untangling a problem.

R Solution

```
# Start with total income
income <- 200000

# x will hold income that hasn't been taxed yet
x <- income
# y will hold taxes paid
y <- 0

if (x <= 10275) {
  y <- x*.1 # tax paid
  x <- 0 # All money has been taxed
} else {
  y <- y + 10275 * .1
  x <- x - 10275 # Money remaining that hasn't been taxed
}

if (x <= (41775 - 10275)) {
  y <- y + x * .12
  x <- 0
} else {
  y <- y + (41775 - 10275) * .12
  x <- x - (41775 - 10275)
}

if (x <= (89075 - 41775)) {
  y <- y + x * .22
  x <- 0
} else {
  y <- y + (89075 - 41775) * .22
  x <- x - (89075 - 41775)
}

if (x <= (170050 - 89075)) {
  y <- y + x * .24
  x <- 0
} else {
  y <- y + (170050 - 89075) * .24
  x <- x - (170050 - 89075)
}

if (x <= (215950 - 170050)) {
  y <- y + x * .32
  x <- 0
} else {
  y <- y + (215950 - 170050) * .32
  x <- x - (215950 - 170050)
}

if (x <= (539900 - 215950)) {
```

Python Solution

```
# Start with total income
income = 200000

# untaxed will hold income that hasn't been taxed yet
untaxed = income
# taxed will hold taxes paid
taxes = 0

if untaxed <= 10275:
  taxes = untaxed*.1 # tax paid
  untaxed = 0 # All money has been taxed
else:
  taxes = taxes + 10275 * .1
  untaxed = untaxed - 10275 # money remaining that hasn't been taxed

if untaxed <= (41775 - 10275):
  taxes = taxes + untaxed * .12
  untaxed = 0
else:
  taxes = taxes + (41775 - 10275) * .12
  untaxed = untaxed - (41775 - 10275)


if untaxed <= (89075 - 41775):
  taxes = taxes + untaxed * .22
  untaxed = 0
else:
  taxes = taxes + (89075 - 41775) * .22
  untaxed = untaxed - (89075 - 41775)

if untaxed <= (170050 - 89075):
  taxes = taxes + untaxed * .24
  untaxed = 0
else:
  taxes = taxes + (170050 - 89075) * .24
  untaxed = untaxed - (170050 - 89075)

if untaxed <= (215950 - 170050):
  taxes = taxes + untaxed * .32
  untaxed = 0
else:
  taxes = taxes + (215950 - 170050) * .32
  untaxed = untaxed - (215950 - 170050)

if untaxed <= (539900 - 215950):
  taxes = taxes + untaxed * .35
  untaxed = 0
else:
  taxes = taxes + (539900 - 215950) * .35
```

We will find a better way to represent this calculation once we discuss loops - we can store each bracket's start and end point in a vector and loop through them. Any time you find yourself copy-pasting code and changing values, you should consider using a loop (or eventually a function) instead.

9.2 Loops

Often, we write programs which update a variable in a way that the new value of the variable depends on the old value:

```
x = x + 1
```

This means that we add one to the current value of `x`.

Before we write a statement like this, we have to **initialize** the value of `x` because otherwise, we don't know what value to add one to.

```
x = 0
x = x + 1
```

We sometimes use the word **increment** to talk about adding one to the value of `x`; **decrement** means subtracting one from the value of `x`.

A particularly powerful tool for making these types of repetitive changes in programming is the **loop**, which executes statements a certain number of times. Loops can be written in several different ways, but all loops allow for executing a block of code a variable number of times.

9.2.1 While Loops

In the previous section, we discussed conditional statements, where a block of code is only executed *if* a logical statement is true. The simplest type of loop is the **while** loop, which executes a block of code until a statement is no longer true.

⚠ Example - While Loops

9.2.1.0.1 Flow Map

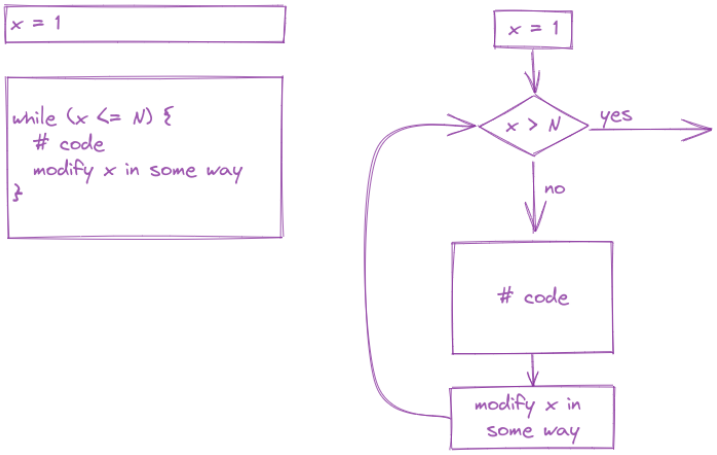


Figure 9.7: Flow map showing while-loop pseudocode (while x <= N) { # code that changes x in some way} and the program flow map expansion where we check if x > N (exiting the loop if true); otherwise, we continue into the loop, execute the main body of #code and then change x and start over.

9.2.1.0.2 * R

```r
x <- 0

while (x < 10) {
  # Everything in here is executed
  # during each iteration of the loop
  print(x)
  x <- x + 1
}
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

9.2.1.0.3 * Python

```python
x = 0

while x < 10:
  print(x)
  x = x + 1
## 0
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
```

Try it Out - While Loops

9.2.1.0.4 * Problem
Write a while loop that verifies that

$$\lim_{N\to\infty} \prod_{k=1}^{N} \left(1 + \frac{1}{k^2}\right) = \frac{e^{\pi} - e^{-\pi}}{2\pi}.$$

Terminate your loop when you get within 0.0001 of $\frac{e^{\pi}-e^{-\pi}}{2\pi}$. At what value of k is this point reached?

9.2.1.0.5 * Math Notation
Breaking down math notation for code:

- If you are unfamiliar with the notation $\prod_{k=1}^{N} f(k)$, this is the product of $f(k)$ for $k = 1, 2, ..., N$,

$$f(1) \cdot f(2) \cdot ... \cdot f(N)$$

- To evaluate a limit, we just keep increasing N until we get arbitrarily close to the right hand side of the equation.

In this problem, we can just keep increasing k and keep track of the cumulative product. So we define `k=1`, `prod = 1`, and `ans` before the loop starts. Then, we loop over `k`, multiplying `prod` by $(1 + 1/k^2)$ and then incrementing k by one each time. At each iteration, we test whether `prod` is close enough to `ans` to stop the loop.

9.2.1.0.6 * R Solution
In R, you will use `pi` and `exp()` - these are available by default without any additional libraries or packages.

```
k <- 1
prod <- 1
ans <- (exp(pi) - exp(-pi))/(2*pi)
delta <- 0.0001

while (abs(prod - ans) >= 0.0001) {
  prod <- prod * (1 + 1/k^2)
  k <- k + 1
}

k
## [1] 36761
prod
## [1] 3.675978
ans
## [1] 3.676078
```

9.2.1.0.7 * Python solution

Note that in python, you will have to import the math library to get the values of pi and the `exp` function. You can refer to these as `math.pi` and `math.exp()` respectively.

```
import math

k = 1
prod = 1
ans = (math.exp(math.pi) - math.exp(-math.pi))/(2*math.pi)
delta = 0.0001

while abs(prod - ans) >= 0.0001:
  prod = prod * (1 + k**-2)
  k = k + 1
  if k > 500000:
    break


print("At ", k, " iterations, the product is ", prod, "compared to the limit ", ans,".")
## At  36761  iterations, the product is  3.675977910975878 compared to the limit  3.67607
```

Warning: Avoid Infinite Loops

It is very easy to create an **infinite** loop when you are working with while loops. Infinite loops never exit, because the condition is always true. If in the while loop example we decrement x instead of incrementing x, the loop will run forever.
You want to try very hard to avoid ever creating an infinite loop - it can cause your session to crash.
One common way to avoid infinite loops is to create a second variable that just counts how many times the loop has run. If that variable gets over a certain threshold, you exit the loop.

9.2.1.0.8 * R

This while loop runs until either x $<$ 10 or n $>$ 50 - so it will run an indeterminate number of times and depends on the random values added to x. Since this process (a 'random walk') could theoretically continue forever, we add the n>50 check to the loop so that we don't tie up the computer for eternity.

```r
x <- 0
n <- 0 # count the number of times the loop runs

while (x < 10) {
  print(x)
  x <- x + rnorm(1) # add a random normal (0, 1) draw each time
  n <- n + 1
  if (n > 50)
    break # this stops the loop if n > 50
}
```

```
## [1] 0
## [1] -0.4953706
## [1] 1.446271
## [1] 2.640349
## [1] 0.09064262
## [1] 0.6050325
## [1] 0.3571766
## [1] -2.048957
## [1] -3.435244
## [1] -4.661981
## [1] -6.122961
## [1] -5.779251
## [1] -7.586729
## [1] -7.710076
## [1] -6.592957
## [1] -6.828925
## [1] -6.770769
## [1] -4.582738
## [1] -5.733723
## [1] -7.135094
## [1] -6.021466
## [1] -6.939712
## [1] -6.358589
## [1] -5.968556
## [1] -6.049961
## [1] -6.982148
## [1] -7.217816
## [1] -6.745364
## [1] -7.874618
## [1] -6.51325
## [1] -6.200296
## [1] -6.383202
## [1] -6.718538
## [1] -5.939521
## [1] -5.037442
## [1] -4.729334
## [1] -5.191015
## [1] -5.525614
## [1] -6.337335
## [1] -7.277082
```

9.2.1.0.9 * Python

```
import numpy as np; # for the random normal draw

x = 0
n = 0 # count the number of times the loop runs

while x < 10:
  print(x)
  x = x + np.random.normal(0, 1, 1) # add a random normal (0, 1) draw each time
  n = n + 1
  if n > 50:
    break # this stops the loop if n > 50
## 0
## [-1.11122275]
## [0.45033693]
## [-0.34511949]
## [-0.57134646]
## [-1.40697539]
## [0.09967624]
## [0.63583808]
## [1.67753754]
## [1.89540881]
## [2.71935926]
## [2.25133996]
## [1.453354]
## [2.2566844]
## [2.51512086]
## [2.70298543]
## [3.38509021]
## [5.93672768]
## [6.34586625]
## [6.33022767]
## [5.97302059]
## [6.59174139]
## [5.64758744]
## [5.61500966]
## [5.51986513]
## [5.58796915]
## [5.51389984]
## [6.0215984]
## [7.31439394]
## [9.37065185]
## [8.78848034]
## [7.60006381]
## [8.07658866]
## [7.29186211]
## [7.96347023]
## [5.81381629]
## [6.59567968]
## [7.80048094]
## [8.03765584]
```

In both of the examples above, there are more efficient ways to write a random walk, but we will get to that later. The important thing here is that we want to make sure that our loops don't run for all eternity.

9.2.2 For Loops

Another common type of loop is a **for** loop. In a for loop, we run the block of code, iterating through a series of values (commonly, one to N, but not always). Generally speaking, for loops are known as **definite** loops because the code inside a for loop is executed a specific number of times. While loops are known as **indefinite** loops because the code within a while loop is evaluated until the condition is falsified, which is not always a known number of times.

Example - For Loop Syntax

9.2.2.0.1 * Flow Map

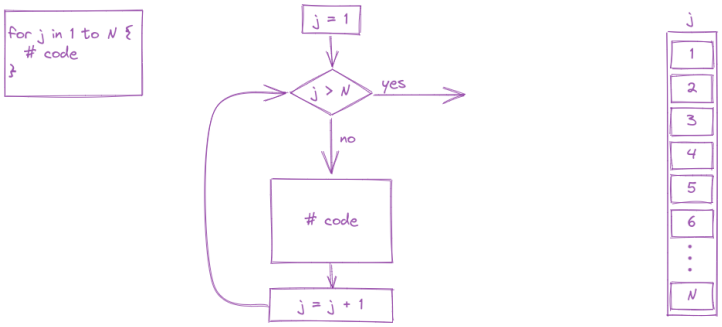


Figure 9.8: Flow map showing for-loop pseudocode (for j in 1 to N) { # code} and the program flow map expansion where j starts at 1 and we check if j > N (exiting the loop if true); otherwise, we continue into the loop, execute the main body of #code and then increment j and start over.

9.2.2.0.2 * R

```
for (i in 1:5 ) {
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

9.2.2.0.3 * Python

```
for i in range(5):
  print(i)
## 0
## 1
## 2
## 3
## 4
```

By default `range`(5) goes from 0 to 5, the upper bound. When `i = 5` the loop exits. This is because `range(5)` creates a vector `[0, 1, 2, 3, 4]`.

For loops are often run from 1 to N (or 0 to N-1 in python) but in essence, a for loop is very commonly used to do a task for every value of a vector.

9.2.2.1 Example - For Loops

9.2.2.1.1 * R

For instance, in R, there is a built-in variable called `month.name`. Type `month.name` into your R console to see what it looks like. If we want to iterate along the values of month.name, we can:

```
for (i in month.name)
  print(i)
```

```
## [1] "January"
## [1] "February"
## [1] "March"
## [1] "April"
## [1] "May"
## [1] "June"
## [1] "July"
## [1] "August"
## [1] "September"
## [1] "October"
## [1] "November"
## [1] "December"
```

9.2.2.1.2 * Python

In python, we have to define our vector or list to start out with, but that's easy enough:

```
futurama_crew = ['Fry', 'Leela', 'Bender', 'Amy', 'the Professor', 'Hermes', 'Zoidberg', 'Ni
for i in futurama_crew:
  print(i)
## Fry
## Leela
## Bender
## Amy
## the Professor
## Hermes
## Zoidberg
## Nibbler
```

9.3 Other Control Structures

9.3.1 Conditional Statements

case statements, e.g. `case_when` in tidyverse

9.3.2 Loops

9.3.2.1 Controlling Loops

While I do not often use break, next, and continue statements, they do exist in both languages and can be useful for controlling the flow of program execution. I have moved the section on this to Section 24.2 for the sake of brevity and to reduce the amount of new material those without programming experience are being exposed to in this section.

9.3.2.2 Other Types of Loops

There are other types of loops in most languages, such as the do-while loop, which runs the code first and then evaluates the logical condition to determine whether the loop will be run again.

⚠ Example: do-while loops

9.3.2.2.0.1 R

In R, do-while loops are most naturally implemented using a very primitive type of iteration: a `repeat` statement.

```
repeat {
  # statements go here
  if (condition)
    break # this exits the repeat statement
}
```

9.3.2.2.0.2 Python

In python, do-while loops are most naturally implemented using a while loop with condition TRUE:

```
while TRUE:
  # statements go here
  if condition:
    break
```

An additional means of running code an indeterminate number of times is the use of **recursion**, which we cannot cover until we learn about functions. I have added an additional section, Section 24.3, to cover this topic, but it is not essential to being able to complete most basic data programming tasks. Recursion is useful when working with structures such as trees (including phylogenetic trees) and nested lists.

9.4 References

10 Writing Functions

A **function** is a set of actions that we group together and name. Throughout this course, you've used a bunch of different functions in R and python that are built into the language or added through packages: `mean`, `ggplot`, `length`, `print`. In this chapter, we'll be writing our own functions.

10.1 Objectives

- Identify the parts of a function from provided source code
- Predict what the function will return when provided with input values and source code
- Given a task, lay out the steps necessary to complete the task in pseudocode
- Write a function which uses necessary input values to complete a task

10.2 When to write a function?

If you've written the same code (with a few minor changes, like variable names) more than twice, you should probably write a function instead. There are a few benefits to this rule:

1. Your code stays neater (and shorter), so it is easier to read, understand, and maintain.
2. If you need to fix the code because of errors, you only have to do it in one place.
3. You can re-use code in other files by keeping functions you need regularly in a file (or if you're really awesome, in your own package!)

4. If you name your functions well, your code becomes easier to understand thanks to grouping a set of actions under a descriptive function name.

Learn more about functions

There is some extensive material on this subject in R for Data Science [14] on functions. If you want to really understand how functions work in R, that is a good place to go.

Example: Turning Code into Functions

This example is modified from R for Data Science [15, Ch. 19].
What does this code do? Does it work as intended?

R

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
## Error in loadNamespace(x): there is no package called 'tibble'

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
## Error in df$a: object of type 'closure' is not subsettable
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
## Error in df$b: object of type 'closure' is not subsettable
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
## Error in df$c: object of type 'closure' is not subsettable
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
## Error in df$d: object of type 'closure' is not subsettable
```

10.2.0.1 Python

```
import pandas as pd
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
import numpy as np

df = pd.DataFrame({
  'a': np.random.randn(10),
  'b': np.random.randn(10),
  'c': np.random.randn(10),
  'd': np.random.randn(10)})
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'pd' is not
df.a = (df.a - min(df.a))/(max(df.a) - min(df.a))
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
df.b = (df.b - min(df.b))/(max(df.b) - min(df.a))
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
df.c = (df.c - min(df.c))/(max(df.c) - min(df.c))
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
df.d = (df.d - min(df.d))/(max(df.d) - min(df.d))
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
```

The code rescales a set of variables to have a range from 0 to 1. But, because of the copy-pasting, the code's author made a mistake and forgot to change an `a` to `b`.
Writing a function to rescale a variable would prevent this type of copy-paste error.
To write a function, we first analyze the code to determine how many inputs it has:

R

```
df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
## Error in df$a: object of type 'closure' is not subsettable
```

This code has only one input: `df$a`.

Python

```
df.a = (df.a - min(df.a))/(max(df.a) - min(df.a))
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
```

This code has only one input: `df.a`
To convert the code into a function, we start by rewriting it using general names:

R

In this case, it might help to replace `df$a` with `x`.

```
x <- df$a
## Error in df$a: object of type 'closure' is not subsettable

(x - min(x, na.rm = TRUE)) /
  (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
## Error in eval(expr, envir, enclos): object 'x' not found
```

Python

In this case, it might help to replace `df.a` with `x`.

```
x = df.a
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
(x - min(x))/(max(x) - min(x))
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'x' is not d
```

Then, we make it a bit easier to read, removing duplicate computations if possible (for instance, computing `min` two times).

R

In R, we can use the `range` function, which computes the maximum and minimum at the same time and returns the result as `c(min, max)`

```r
rng <- range(x, na.rm = T)
## Error in eval(expr, envir, enclos): object 'x' not found

(x - rng[1])/(rng[2] - rng[1])
## Error in eval(expr, envir, enclos): object 'x' not found
```

Python

In python, `range` is the equivalent of `seq()` in R, so we are better off just using `min` and `max`.

```r
x = df.a

## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
xmin, xmax = [x.min(), x.max()]
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'x' is not d
(x - xmin)/(xmax - xmin)
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'x' is not d
```

Finally, we turn this code into a function:

R

```r
rescale01 <- function(x) {
  rng <- range(x, na.rm = T)
  (x - rng[1])/(rng[2] - rng[1])
}

rescale01(df$a)
## Error in df$a: object of type 'closure' is not subsettable
```

- The name of the function, `rescale01`, describes what the function does - it rescales the data to between 0 and 1.
- The function takes one **argument**, named `x`; any references to this value within the function will use `x` as the name. This allows us to use the function on `df$a`, `df$b`, `df$c`, and so on, with `x` as a placeholder name for the data we're working on at the moment.

- The code that actually does what your function is supposed to do goes in the **body** of the function, between `{` and `}` (this is true in R, in python, there are different conventions, but the same principle applies)
- The function **returns** the last value computed: in this case, `(x - rng[1])/(rng[2]-rng[1])`. You can make this explicit by adding a `return()` statement around that calculation.

Python

```
def rescale01(x):
  xmin, xmax = [x.min(), x.max()]
  return (x - xmin)/(xmax - xmin)

rescale01(df.a)
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'df' is not
```

- The name of the function, `rescale01`, describes what the function does - it rescales the data to between 0 and 1.
- The function takes one **argument**, named `x`; any references to this value within the function will use `x` as the name. This allows us to use the function on `df.a`, `df.b`, `df.c`, and so on, with `x` as a placeholder name for the data we're working on at the moment.
- The code that actually does what your function is supposed to do goes in the **body** of the function, indented relative to the line with `def function_name():`. At the end of the function, you should have a blank line with no spaces or tabs.
- The function **returns** the value it is told to `return`: in this case, `(x - xmin)/(xmax - xmin)`. In Python, you must `return` a value if you want the function to perform a computation. [a]

The process for creating a function is important: first, you figure out how to do the thing you want to do. Then, you simplify the code as much as possible. Only at the end of that process do you create an actual function.

[a]This is not strictly true, you can of course use pass-by-reference, but we will not be covering that in this class as we are strictly dealing with the bare minimum of learning how to write a function here.

10.3 Syntax

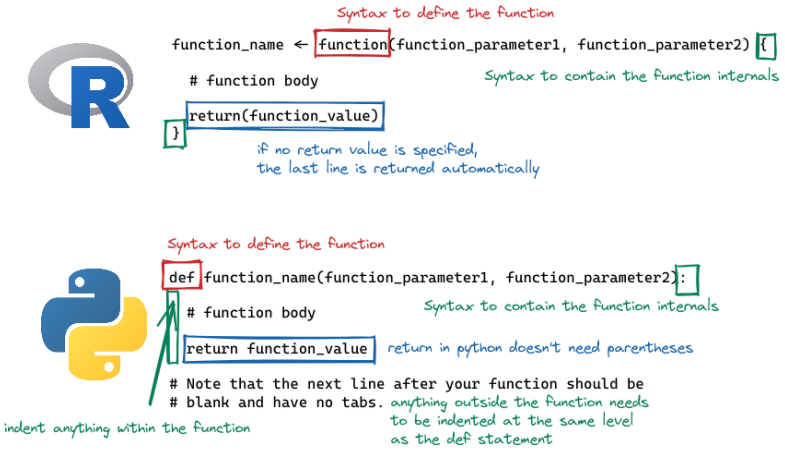


Figure 10.1: R and python syntax for defining functions. Portions of the command that indicate the function name, function scope, and return statement are highlighted in each block.

In R, functions are defined as other variables, using `<-`, but we specify the arguments a function takes by using the `function()` statement. The contents of the function are contained within `{` and `}`. If the function returns a value, a `return()` statement can be used; alternately, if there is no return statement, the last computation in the function will be returned.

In python, functions are defined using the `def` command, with the function name, parentheses, and the function arguments

to follow. The first line of the function definition ends with a `:`, and all subsequent lines of the function are indented (this is how python knows where the end of the function is). A python function return statement is `return <value>`, with no parentheses needed.

Note that in python, the `return` statement is not optional. It is not uncommon to have python functions that don't return anything; in R, this is a bit less common, for reasons we won't get into here.

10.4 Arguments and Parameters

An **argument** is the name for the object you pass into a function.

A **parameter** is the name for the object once it is inside the function (or the name of the thing as defined in the function).

Example: Parts of a Function

Let's examine the difference between arguments and parameters by writing a function that takes a dog's name and returns " is a good pup!".

R

```r
dog <- "Eddie"

goodpup <- function(name) {
  paste(name, "is a good pup!")
}

goodpup(dog)
## [1] "Eddie is a good pup!"
```

Python

```
dog = "Eddie"

def goodpup(name):
  return name + " is a good pup!"

goodpup(dog)
## 'Eddie is a good pup!'
```

In this example function, when we call `goodpup(dog)`, `dog` is the argument. `name` is the parameter.
What is happening inside the computer's memory as `goodpup` runs?

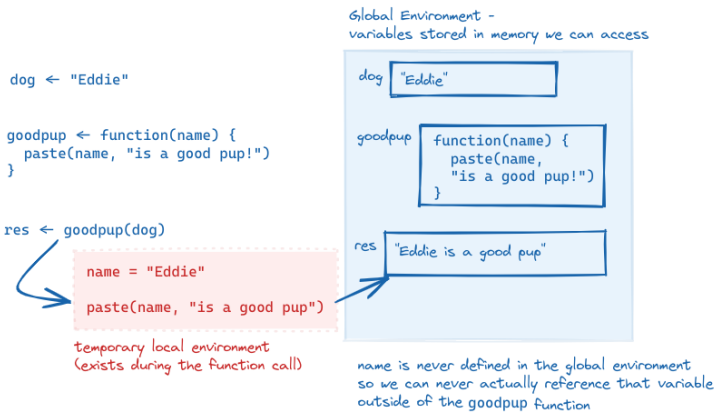


Figure 10.2: A sketch of the execution of the program `goodpup`, showing that `name` is only defined within the local environment that is created while `goodpup` is running. We can never access `name` in our global environment.

This is why the distinction between **arguments** and **parameters** matters. Parameters are only accessible while inside of the function - and in that local environment, we need to call the object by the parameter name, not the name we use outside the function (the argument name).

We can even call a function with an argument that isn't de-

fined outside of the function call: `goodpup("Tesla")` produces "Tesla is a good pup!". Here, I do not have a variable storing the string "Tesla", but I can make the function run anyways. So "Tesla" here is an argument to `goodpup` but it is not a variable in my environment.

This is a confusing set of concepts and it's ok if you only just sort of get what I'm trying to explain here. Hopefully it will become more clear as you write more code.

Try it out: Function Parts

For each of the following blocks of code, identify the function name, function arguments, parameter names, and return statements. When the function is called, see if you can predict what the output will be. Also determine whether the function output is stored in memory or just printed to the command line.

Function 1

```
def hello_world():
  print("Hello World")
```

```
hello_world()
```

Answer

- Function name: `hello_world`
- Function parameters: none
- Function arguments: none
- Function output:

```
hello_world()
## Hello World
```

- Function output is not stored in memory and is printed to the command line.

Function 2

```
my_mean <- function(x) {
  censor_x <- sample(x, size = length(x) - 2, replace = F)
  mean(censor_x)
}
```

```
set.seed(3420523)
x = my_mean(1:10)
x
```

Answer

- Function name: `my_mean`
- Function parameters: x
- Function arguments: 1:10
- Function output: (varies each time the function is run unless you set the seed)

```
set.seed(3420523)
x = my_mean(1:10)
x
## [1] 6
```

- Function output is saved to memory (`x`) and printed to the command line

10.4.1 Named Arguments and Parameter Order

In the examples above, you didn't have to worry about what order parameters were passed into the function, because there were 0 and 1 parameters, respectively. But what happens when we have a function with multiple parameters?

R

```r
divide <- function(x, y) {
  x / y
}
```

Python

```python
def divide(x, y):
  return x / y
```

In this function, the order of the parameters matters! `divide(3, 6)` does not produce the same result as `divide(6, 3)`. As you might imagine, this can quickly get confusing as the number of parameters in the function increases.

In this case, it can be simpler to use the parameter names when you pass in arguments.

R

```r
divide(3, 6)
## [1] 0.5

divide(x = 3, y = 6)
## [1] 0.5

divide(y = 6, x = 3)
## [1] 0.5

divide(6, 3)
## [1] 2

divide(x = 6, y = 3)
## [1] 2

divide(y = 3, x = 6)
```

```
## [1] 2
```

Python

```
divide(3, 6)
## 0.5
divide(x = 3, y = 6)
## 0.5
divide(y = 6, x = 3)
## 0.5
divide(6, 3)
## 2.0
divide(x = 6, y = 3)
## 2.0
divide(y = 3, x = 6)
## 2.0
```

As you can see, the order of the arguments doesn't much matter, as long as you use named arguments, but if you don't name your arguments, the order very much matters.

10.4.2 Input Validation

When you write a function, you often assume that your parameters will be of a certain type. But you can't guarantee that the person using your function knows that they need a certain type of input. In these cases, it's best to **validate** your function input.

Input Validation Example

10.4.2.0.1 * R

In R, you can use `stopifnot()` to check for certain essential conditions. If you want to provide a more illuminating error message, you can check your conditions using `if()` and then use `stop("better error message")` in the body of the if statement.

```r
add <- function(x, y) {
  x + y
}

add("tmp", 3)
## Error in x + y: non-numeric argument to binary operator

add <- function(x, y) {
  stopifnot(is.numeric(x))
  stopifnot(is.numeric(y))
  x + y
}

add("tmp", 3)
## Error in add("tmp", 3): is.numeric(x) is not TRUE
add(3, 4)
## [1] 7
```

10.4.2.0.2 * Python

In Python, the easiest way to handle errors is to use a try statement, which operates rather like an if statement: if the statement executes, then we're good to go; if not, we can use `except` to handle different types of errors. The `else` clause is there to handle anything that needs to happen if the statement in the `try` clause executes without any errors.

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

add("tmp", 3)
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: can only concaten
def add(x, y):
  try:
    return x + y
  except TypeError:
    print("x and y must be add-able")
  else:
    # We should never get here, because the try clause has a return statement
    print("Else clause?")
  return

add("tmp", 3)
## x and y must be add-able
add(3, 4)
## 7
```

You can read more about error handling in Python here

Input validation is one aspect of **defensive programming** - programming in such a way that you try to ensure that your programs don't error out due to unexpected bugs by anticipating ways your programs might be misunderstood or misused [16].

10.5 Scope

When talking about functions, for the first time we start to confront a critical concept in programming, which is scope. **Scope** is the part of the program where the name you've given a variable is valid - that is, where you can use a variable.

> A variable is only available from inside the region it is created.

What do I mean by the part of a program? The **lexical scope** is the portion of the code (the set of lines of code) where the name is valid.

The concept of scope is best demonstrated through a series of examples, so in the rest of this section, I'll show you some examples of how scope works and the concepts that help you figure out what "scope" actually means in practice.

10.5.1 Name Masking

Scope is most clearly demonstrated when we use the same variable name inside and outside a function. Note that this is 1) bad programming practice, and 2) fairly easily avoided if you can make your names even slightly more creative than `a`, `b`, and so on. But, for the purposes of demonstration, I hope you'll forgive my lack of creativity in this area so that you can see how name masking works.

> Danger
>
> What does this function return, 10 or 20?
>
> **Pseudocode**

```
a = 10

myfun = function() {
  a = 20
  return a
}

myfun()
```

Sketch

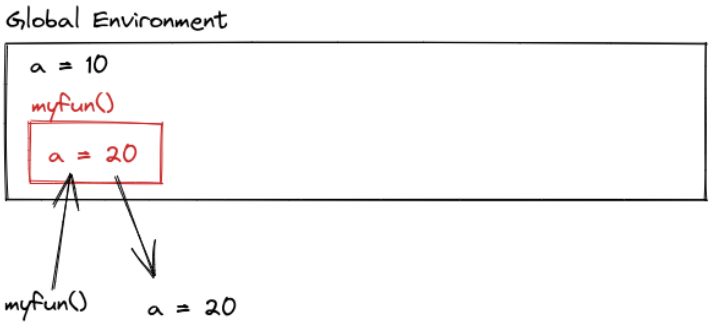


Figure 10.3: A sketch of the global environment as well as the environment within `myfun()`. Because `a=20` inside `myfun()`, when we call `myfun()`, we get the value of `a` within that environment, instead of within the global environment.

R

```
a <- 10

myfun <- function() {
  a <- 20
  a
}

myfun()
## [1] 20
```

Python

```
a = 10

def myfun():
  a = 20
  return a

myfun()
## 20
```

The lexical scope of the function is the area that is between the braces (in R) or the indented region (in python). Outside the function, `a` has the value of 10, but inside the function, `a` has the value of 20. So when we call `myfun()`, we get 20, because the scope of `myfun` is the **local context** where `a` is evaluated, and the value of `a` in that environment dominates.

This is an example of **name masking**, where names defined inside of a function mask names defined outside of a function.

10.5.2 Environments and Scope

Another principle of scoping is that if you call a function and then call the same function again, the function's environment is re-created each time. Each function call is unrelated to the next function call when the function is defined using local variables.

Danger

Pseudocode

```
myfun = function() {
  if a is not defined
    a = 1
  else
    a = a + 1
}
```

```
myfun()
myfun()
```

What does this output?

Sketch

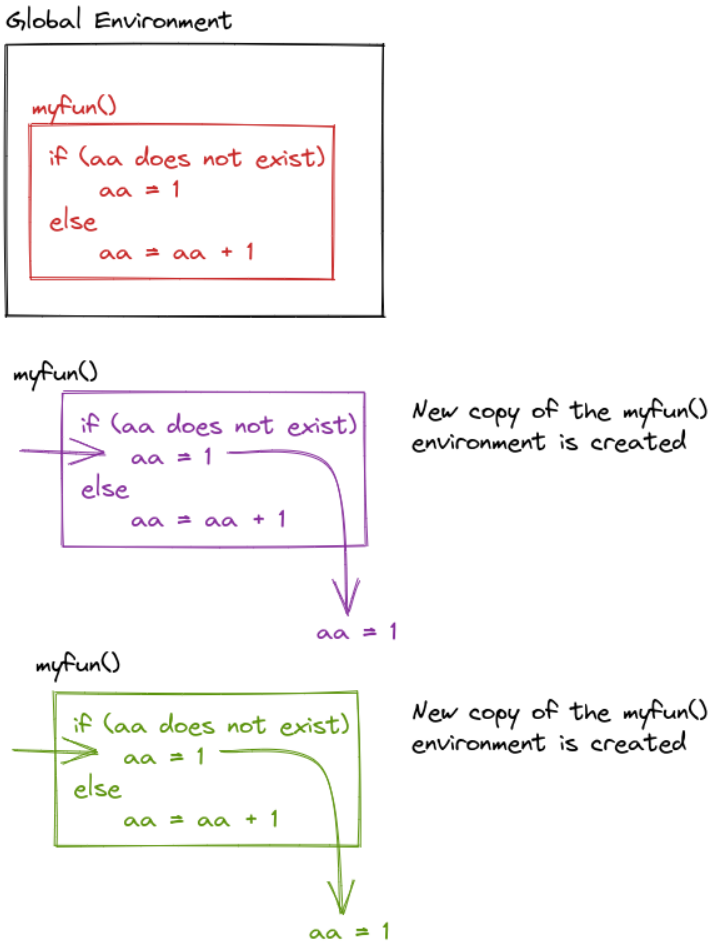


Figure 10.4: When we define myfun, we create a template for an environment with variables and code to excecute. Each time `myfun()` is called, that template is used to create a new environment. This prevents successive calls to `myfun()` from affecting each other – which means `a = 1` every time.

R

```
myfun <- function() {
  if (!exists("aa")) {
    aa <- 1
  } else {
    aa <- aa + 1
  }
  return(aa)
}

myfun()
## [1] 1
myfun()
## [1] 1
```

Python

```
def myfun():
  try: aa
  except NameError: aa = 1
  else: aa = aa + 1
  return aa

myfun()
## 1
myfun()
## 1
```

Note that the `try` command here is used to handle the case where a doesn't exist. If there is a NameError (which will happen if `aa` is not defined) then we define `aa = 1`, if there is not a NameError, then `aa = aa + 1`.
This is necessary because Python does not have a built-in way to test if a variable exists before it is used [17], Ch 17.

10.5.3 Dynamic Lookup

Scoping determines where to look for values – when, however, is determined by the sequence of steps in the code. When a function is called, the **calling environment** (the global environment or set of environments at the time the function is called) determines what values are used.

If an object doesn't exist in the function's environment, the global environment will be searched next; if there is no object in the global environment, the program will error out. This behavior, combined with changes in the calling environment over time, can mean that the output of a function can change based on objects outside of the function.

Danger

Pseudocode

```
myfun = function() x + 1

x = 14

myfun()

x = 20

myfun()
```

What will the output be of this code?

Sketch

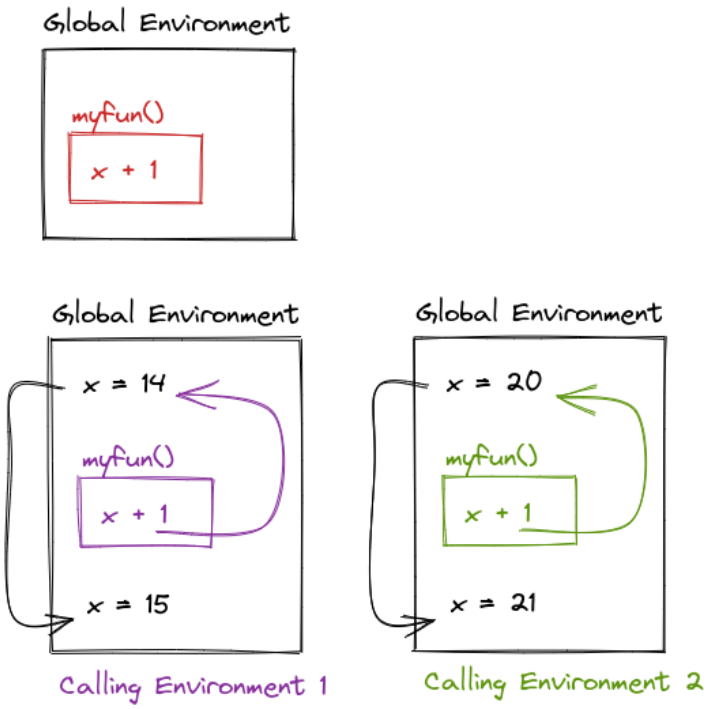


Figure 10.5: The state of the global environment at the time the function is called (that is, the state of the calling environment) can change the results of the function

R

```r
myfun <- function() {
  x + 1
}

x <- 14

myfun()
## [1] 15

x <- 20

myfun()
## [1] 21
```

Python

```python
def myfun():
  return x + 1


x = 14

myfun()
## 15
x = 20

myfun()
## 21
```

Try It Out: Function Scope

What does the following function return? Make a prediction, then run the code yourself. From [15, Ch. 6]

10.5.3.0.1 * R code

```r
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x ^ 2
    }
    f() + 1
  }
  f(x) * 2
}
f(10)
```

10.5.3.0.2 * R solution

```r
f <- function(x) {
  f <- function(x) {
    f <- function() {
      x ^ 2
    }
    f() + 1
  }
  f(x) * 2
}
f(10)
## [1] 202
```

10.5.3.0.3 * Python code

```python
def f(x):
  def f(x):
    def f():
      return x ^ 2
    return f() + 1
  return f(x) * 2

f(10)
```

10.5.3.0.4 * Python solution

```
def f(x):
  def f(x):
    def f():
      return x ** 2
    return f() + 1
  return f(x) * 2

f(10)
## 202
```

10.6 References

11 Debugging

Now that you're writing functions, it's time to talk a bit about debugging techniques. This is a lifelong topic - as you become a more advanced programmer, you will need to develop more advanced debugging skills as well (because you'll become more adept at screwing things up).

11.1 Objectives

- Create reproducible examples of problems
- Use built in debugging tools to trace errors
- Use online resources to research errors

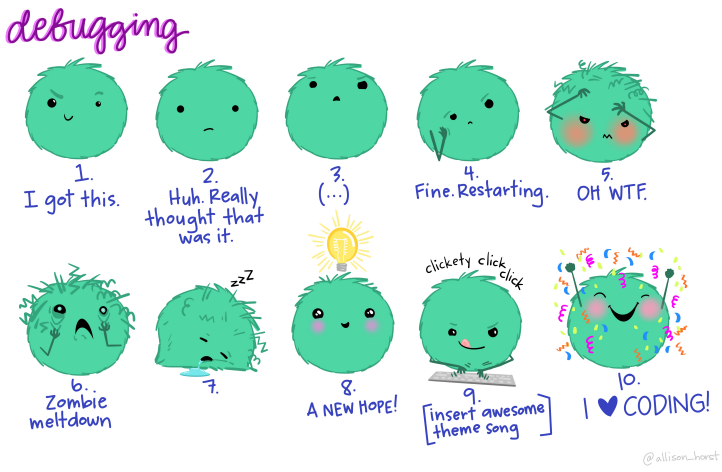


Figure 11.1: The faces of debugging (by Allison Horst)

11.2 Avoiding Errors: Defensive Programming

One of the best debugging strategies (that isn't a debugging strategy at all, really) is to code defensively [16]. By that, I mean, code in a way that you will make debugging things easier later.

- **Modularize your code**. Each function should do only one task, ideally in the least-complex way possible.
- **Make your code readable**. If you can read the code easily, you'll be able to narrow down the location of the bug more quickly.
- **Comment your code**. This makes it more likely that you will be able to locate the spot where the bug is likely to have occurred, and will remind you how things are calculated. Remember, comments aren't just for your collaborators or others who see the code. They're for future you.
- **Don't duplicate code**. If you have the same code (or essentially the same code) in two or three different places, put that code in a function and call the function instead. This will save you trouble when updating the code in the future, but also makes narrowing down the source of the bug less complex.
- **Reduce the number of dependencies** you have on outside software packages. Often bugs are introduced when a dependency is updated and the functionality changes slightly. The `tidyverse` [18] is *notorious* for this.

> **Note**
>
> It's ok to write code using lots of dependencies, but as you transition from "experimental" code to "production" code (you're using the code without tinkering with it) you should work to reduce the dependencies, where possible. In addition, if you do need packages with lots of dependen-

> cies, try to make sure those packages are relatively popular, used by a lot of people, and currently maintained. (The tidyverse is a bit better from this perspective, because the constituent packages are some of the most installed R packages on CRAN.)

Another way to handle dependency management is to use the `renv` package [19], which creates a local package library with the appropriate versions of your packages stored in the same directory as your project. `renv` was inspired by the python concept of virtual environments, and it does also work with python if you're using both R and python inside a project (e.g. this book uses `renv`). `renv` will at the very least help you minimize issues with code not working after unintentional package updates.

- **Add safeguards against unexpected inputs**. Check to make sure inputs to the function are valid. Check to make sure intermediate results are reasonable (e.g. you don't compute the derivative of a function and come up with "a".)
- **Don't reinvent the wheel**. If you have working, tested code for a task, use that! If someone else has working code that's used by the community, don't write your own unless you have a very good reason. The implementation of `lm` has been better tested than your homegrown linear regression.
- Collect your often-reused code in packages that you can easily load and make available to "future you". The process of making a package often encourages you to document your code better than you would a script. A good resource for getting started making R packages is [20], and a similar python book is [21].

11.3 Working through Errors

11.3.1 First steps

11.3.1.1 Get into the right mindset

You can't debug something effectively if you're upset. You have to be in a puzzle-solving, detective mindset to actually solve a problem. If you're already stressed out, try to relieve that stress before you tackle the problem: take a shower, go for a walk, pet a puppy.

11.3.1.2 Check your spelling

I'll guess that 80% of my personal debugging comes down to spelling errors and misplaced punctuation.

11.3.2 General Debugging Strategies

Debugging: Being the detective in a crime movie where you are also the murderer. - some t-shirt I saw once

While defensive programming is a nice idea, if you're already at the point where you have an error you can't diagnose, then... it doesn't help that much. At that point, you'll need some general debugging strategies to work with. The overall process is well described in [15]; I've added some steps that are commonly overlooked and modified the context from the original package development to introductory programming. I've also integrated some lovely illustrations from Julia Evans (@b0rk) to lighten the mood.

0. Realize that you have a bug
1. Read the error message
2. **Google!** Seriously, just google the whole error message. In R you can automate this with the `errorist` and `searcher` packages. Python is so commonly used that you'll likely be able to find help for your issue if you are specific enough.

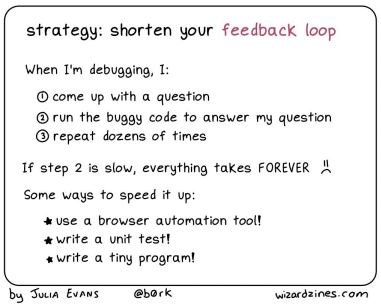


Figure 11.5: Debugging strategy: Shorten your feedback loop [25]

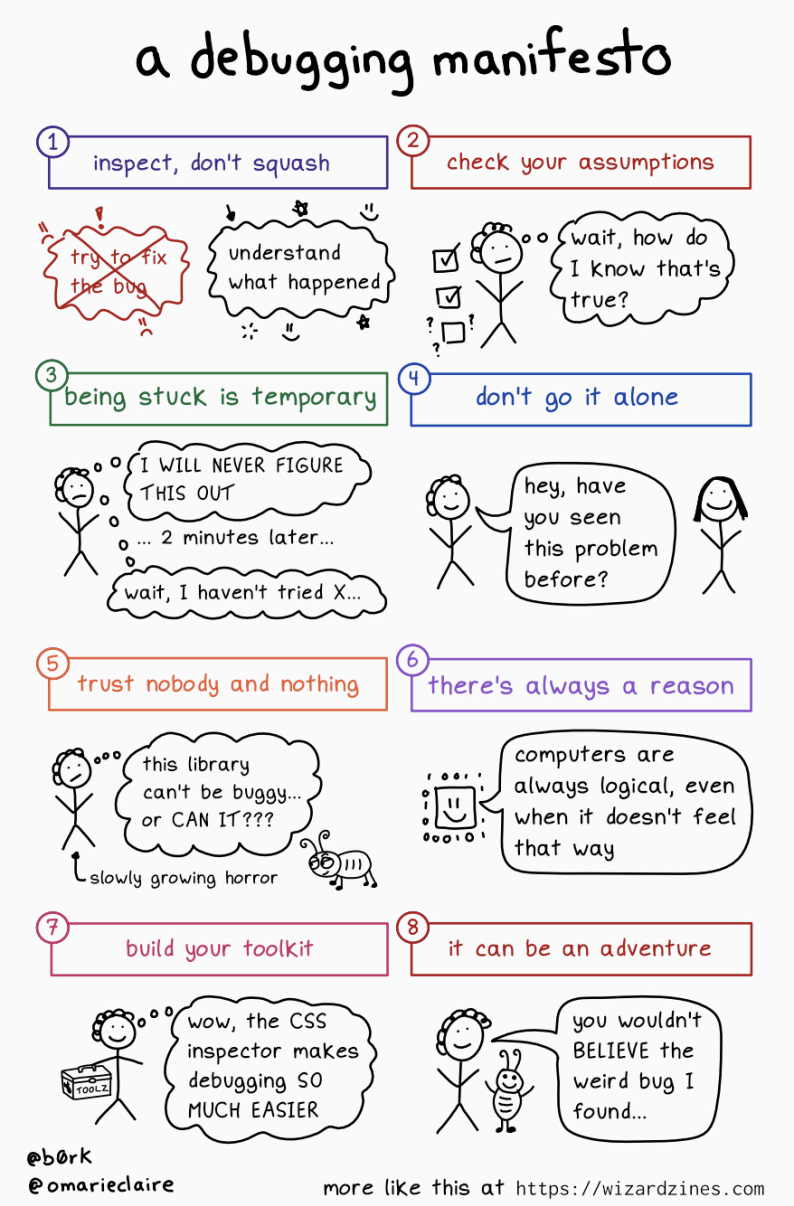


Figure 11.2: A debugging manifesto [22]

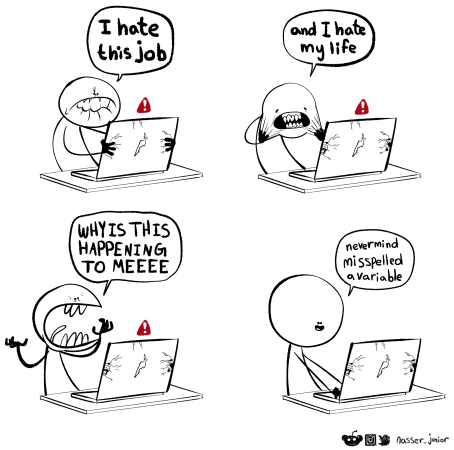


Figure 11.3: Title: user.fist_name [23]

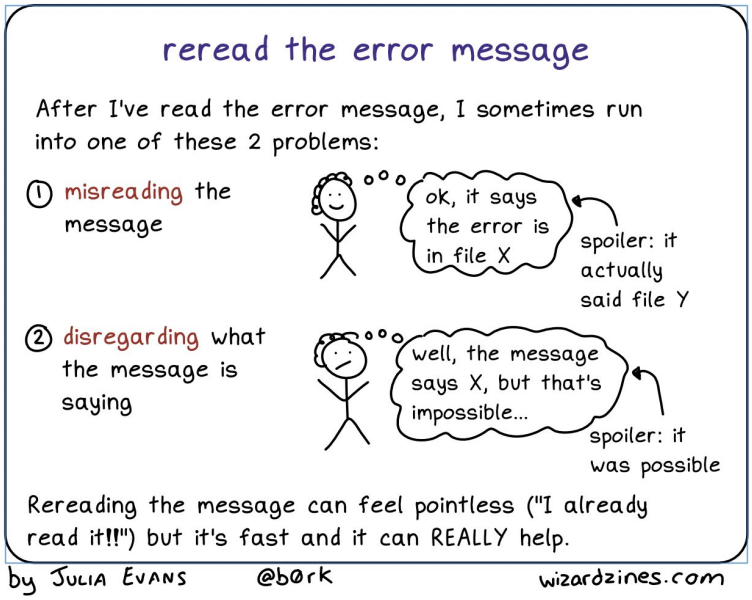
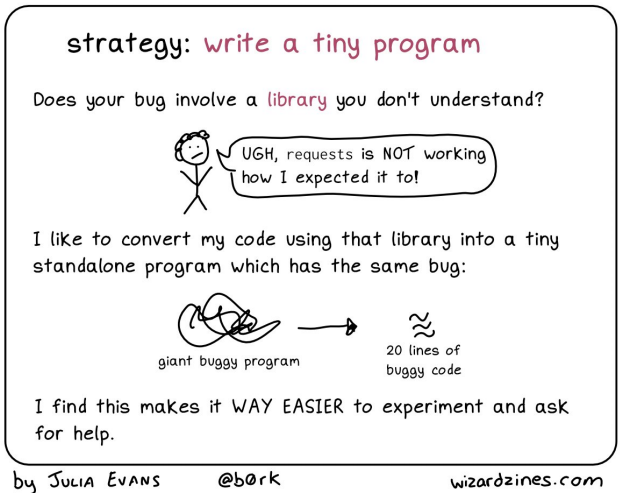


Figure 11.4: Debugging strategy: Reread the error message[24]

3. **Make the error repeatable**: This makes it easier to figure out what the error is, faster to iterate, and easier to ask for help.

    - Use binary search (remove 1/2 of the code, see if the error occurs, if not go to the other 1/2 of the code. Repeat until you've isolated the error.)
    - Generate the error faster - use a minimal test dataset, if possible, so that you can ask for help easily and run code faster. This is worth the investment if you've been debugging the same error for a while.



    - Note which inputs *don't* generate the bug – this negative "data" is helpful when asking for help.

4. **Figure out where it is**. Debuggers may help with this, but you can also use the scientific method to explore the code, or the tried-and-true method of using lots of `print()` statements.

5. **Come up with one question**. If you're stuck, it can be helpful to break it down a bit and ask one tiny question about the bug.

6. **Fix it and test it**. The goal with tests is to ensure that the same error doesn't pop back up in a future version of your code. Generate an example that will test for the

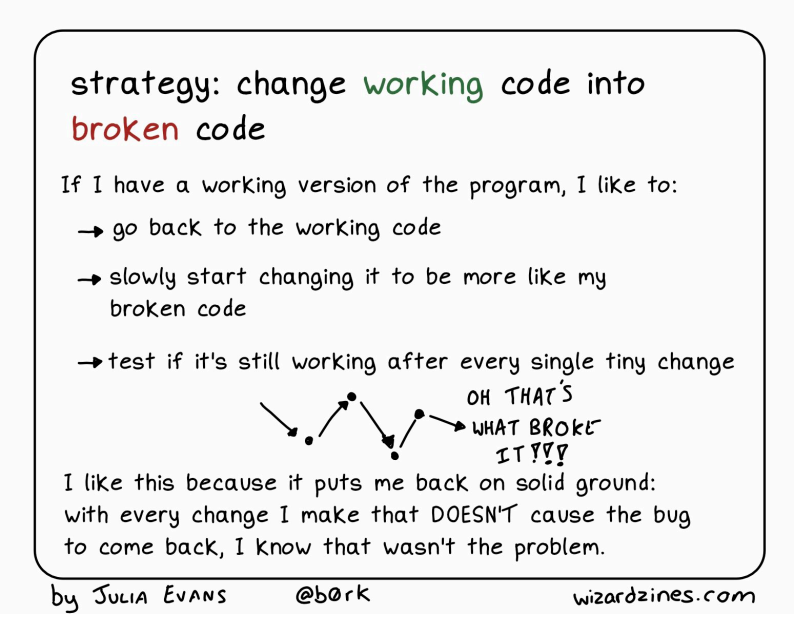


Figure 11.6: Debugging strategy: Change working code into broken code [27]

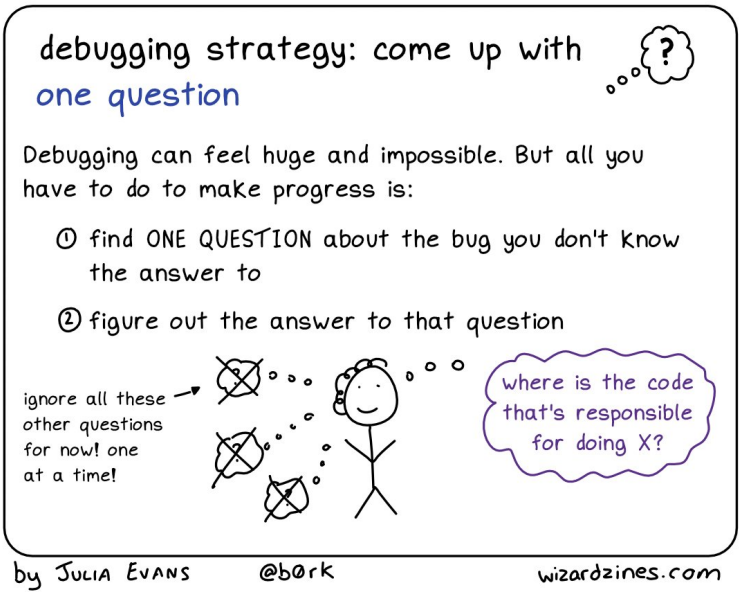


Figure 11.7: Debugging strategy: Come up with one question [28]

error, and add it to your documentation. If you're developing a package, unit test suites offer a more formalized way to test errors and you can automate your testing so that every time your code is changed, tests are run and checked.

There are several other general strategies for debugging:

- Retype (from scratch) your code
  This works well if it's a short function or a couple of lines of code, but it's less useful if you have a big script full of code to debug. However, it does sometimes fix really silly typos that are hard to spot, like having typed `<--` instead of `<-` in R and then wondering why your answers are negative.
- Visualize your data as it moves through the program. This may be done using `print()` statements, or the debugger, or some other strategy depending on your application.
- Tracing statements. Again, this is part of `print()` debugging, but these messages indicate progress - "got into function x", "returning from function y", and so on.
- Rubber ducking. Have you ever tried to explain a problem you're having to someone else, only to have a moment of insight and "oh, never mind"? Rubber ducking outsources the problem to a nonjudgmental entity, such as a rubber duck[1]. You simply explain, in terms simple enough for your rubber duck to understand, exactly what your code does, line by line, until you've found the problem. See [30] for a more thorough explanation.

Do not be surprised if, in the process of debugging, you encounter new bugs. This is a problem that's well-known enough that it has its own xkcd comic. At some point, getting up and

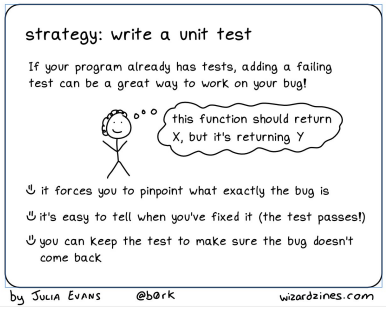


Figure 11.8: Debugging strategy: Write a unit test [29]

[1] Some people use cats, but I find that they don't meet the nonjudgmental criteria. Of course, they're equally judgmental whether your code works or not, so maybe that works if you're a cat person, which I am not. Dogs, in my experience, can work, but often will try to comfort you when they realize you're upset, which both helps and lessens your motivation to fix the problem. A rubber duck is the perfect dispassionate listener.

going for a walk may help. Redesigning your code to be more modular and more organized is also a good idea.

11.4 Dividing Problems into Smaller Parts

> "Divide each difficulty into as many parts as is feasible and necessary to resolve it." -René Descartes, Discourse on Method

In programming, as in life, big, general problems are very hard to solve effectively. Instead, the goal is to break a problem down into smaller pieces that may actually be solvable.

Example: Exhaustion

This example inspired by [31].

11.4.0.1 General problem

"I'm exhausted all the time"
Ok, so this is a problem that many of us have from time to time (or all the time). If we get a little bit more specific at outlining the problem, though, we can sometimes get a bit more insight into how to solve it.

11.4.0.2 Specific problem

"I wake up in the morning and I don't have any energy to do anything. I want to go back to sleep, but I have too much to do to actually give in and sleep. I spend my days worrying about how I'm going to get all of the things on my to-do list done, and then I lie awake at night thinking about how many things there are to do tomorrow. I don't have time for hobbies or exercise, so I drink a lot of coffee instead to make it through the day."
This is a much more specific list of issues, and some of these issues are actually things we can approach separately.

11.4.0.3 Subproblems

Moving through the list in the previous tab, we can isolate a few issues. Some of these issues are undoubtedly related to each other, but we can approach them separately (for the most part).

1. Poor quality sleep (tired in the morning, lying awake at night)
2. Too many things to do (to-do list)
3. Chemical solutions to low energy (coffee during the day)
4. Anxiety about completing tasks (worrying, insomnia)
5. Lack of personal time for hobbies or exercise

11.4.0.4 Brainstorm

1. Get a check-up to rule out any other issues that could cause sleep quality degradation - depression, anxiety, sleep apnea, thyroid conditions, etc.
    - Ask the doctor about taking melatonin supplements for a short time to ensure that sleep starts off well (note, don't take medical advice from a stats textbook!)
2. Reformat your to-do list:
    - Set time limits for things on the to-do list
    - Break the to-do list into smaller, manageable tasks that can be accomplished within a relatively short interval - such as an hour
    - Sort the to-do list by priority and level of "fun" so that each day has a few hard tasks and a couple of easy/fun tasks. Do the hard tasks first, and use the easy/fun tasks as a reward.
3. Set a time limit for caffeine (e.g. no coffee after noon) so that caffeine doesn't contribute to poor quality sleep

4. Address anxiety with medication (from 1), scheduled time for mindfulness meditation, and/or self-care activities
5. Scheduling time for exercise/hobbies

    - scheduling exercise in the morning to take advantage of the endorphins generated by working out
    - scheduling hobbies in the evening to reward yourself for a day's work and wind down work well before bedtime

11.4.0.5 Subproblem solutions

When the sub-problem has a viable solution, move on to the next sub-problem. Don't try to tackle everything at once. Here, that might look like this list, where each step is taken separately and you give each thing a few days to see how it affects your sleep quality. In programming, of course, this list would perhaps be a bit more sequential, but real life is messy and the results take a while to populate.

- [1] Make the doctor's appointment.
- [5] While waiting for the appointment, schedule exercise early in the day and hobbies later in the day to create a "no-work" period before bedtime.
- [1] Go to the doctor's appointment, follow up with any concerns.

    – [1] If doctor approves, start taking melatonin according to directions

- [2] Work on reformatting the to-do list into manageable chunks. Schedule time to complete chunks using your favorite planning method.
- [4] If anxiety is still an issue after following up with the doctor, add some mindfullness meditation or self-care to the schedule in the mornings or evenings.
- [3] If sleep quality is still an issue, set a time limit for caffeine

- [2] Revise your to-do list and try a different tactic if what you were trying didn't work.

11.5 Minimal Working (or Reproducible) Examples

Figure 11.9: The reprex R package will help you make a reproducible example (drawing by Allison Horst)

If all else has failed, and you can't figure out what is causing your error, it's probably time to ask for help. If you have a friend or buddy that knows the language you're working in, by all means ask for help sooner - use them as a rubber duck if you have to. But when you ask for help online, often you're asking people who are much more knowledgeable about the topic - members of R core and really famous python developers browse stackoverflow and may drop in and help you out. Under those circumstances, it's better to make the task of helping you as easy as possible because it shows respect for their time. The same thing goes for your supervisors and professors.

There are numerous resources for writing what's called a "minimal working example", "reproducible example" (commonly abbreviated reprex), or MCVE (minimal complete verifiable example). Much of this is lifted directly from the StackOverflow post describing a minimal reproducible example.

The goal is to reproduce the error message with information that is

- **minimal** - as little code as possible to still reproduce the problem
- **complete** - everything necessary to reproduce the issue is contained in the description/question
- **reproducible** - test the code you provide to reproduce the problem.

You should format your question to make it as easy as possible to help you. Make it so that code can be copied from your post directly and pasted into a terminal. Describe what you see and what you'd hope to see if the code were working.

Other Minimum Working Example/Reprex resources

- reprex package: Do's and Don'ts
- How to use the reprex package - vignette with videos from Jenny Bryan
- reprex magic - Vignette adapted from a blog post by Nick Tierney

Example: MWEs

Note: You don't need to know anything about SAS to understand this example.

11.5.0.1 SAS markdown

A long time ago, when this book covered R and SAS, I had issues with SAS graphs rendering in black and white most of the time.
I started debugging the issue with the following code chunk:
"'{{r sas-cat-aes-map-07, engine="sashtml", engine.path="sas", fig.path = "image/"}} libname classdat "sas/";
PROC SGPLOT data=classdat.fbiwide; SCATTER x = Population y = Assault / markerattrs=(size=8pt symbol=circlefilled) group = Abb; /* maps to point color by default */ RUN; QUIT;
PROC SGPLOT data=classdat.fbiwide NOAUTOLEGEND; /* dont generate a legend / *SCATTER x = Population y = Assault / markercharattrs=(size=8) markerchar = Abb* / specify marker character variable */ group = Abb ; RUN; QUIT;

```
After running the code separately in SAS and getting a figure that looked like what I'd expec

The first thing I did was strip out all of the extra stuff that didn't need to be in the chun
This chunk requires the fbiwide data from the `classdata` R package (that I exported to CSV);
```

When I was done, the chunk looked like this:

PROC SGPLOT data=sashelp.snacks; SCATTER x = date y = QtySold / markerattrs=(size=8pt symbol=circlefilled) group = product; /* maps to point color by default */ RUN; QUIT;

Then, I started constructing my reproducible example.
I ran `?sas_enginesetup` to get to a SASmarkdown help page, because I remembered it had a nic

I copied the example from that page:

11.6 indoc <- '

title: "Basic SASmarkdown Doc" author: "Doug Hemken" output: html_document —

12 I've deleted the intermediate chunks because they screw

13 everything up when I print this chunk out

’ knitr::knit(text=indoc, output=“test.md”) rmarkdown::render(“test.md”)

```
Then, I created several chunks which would do the following:
1. Write the minimal example SAS code above to a [file](files/reprex.sas)
2. Call that file in a SASmarkdown chunk using the `%include` macro, which dumps the listed f
3. Call the file using SAS batch mode
(this runs the code and produces a [plot](files/SASmarkdown-reprex/SGPlot.png) outside of SAS

Finally, I included the image generated from the batch mode call manually.

You can see the resulting code [here](https://github.com/Hemken/SASmarkdown/issues/14).

I pasted my example into the issues page, and then included some additional information:

1. A screenshot of the rendered page
2. The image files themselves
3. A description of what happened
4. My suspicions (some obvious option I'm missing?)
5. An additional line of R code that would delete any files created if someone ran my example

This process took me about 45 minutes, but that was still much shorter than the time I'd sper

In less than 24 hours, the package maintainer responded with a (admittedly terse) explanation
I had to do some additional research to figure out what that meant, but once I had my reprodu

Then, I had to tinker with the book a bit to figure out if there were easier ways to get the
The end result, though, was that I got what I wanted - color figures throughout the book!

#### Python/Quarto
```

```
While converting the book from Rmarkdown to quarto, I ran into an issue setting up GitHub Act


I found an issue describing the same segfault issue I had been getting, and so I made a [post

Within 24h, I had gotten replies from people working at RStudio, and one of them had [diagnos
After I asked a few more questions, one of them submitted a pull request to my repository wit

I didn't know enough python or enough about GitHub Actions to diagnose the problem myself, bu

:::

:::

::: callout-tip
### Try It Out

Use [this list of StackOverflow posts](files/Debugging_exercise.html) to try out your new deb
Can you figure out what's wrong?
What information would you need from the poster in order to come up with a solution?
How much time did you spend trying to figure out what the poster was actually asking?
:::



## Debugging Tools

Now that we've discussed general strategies for debugging that will work in any language, let

### Low tech debugging with print() and other tools
Sometimes called "tracing" techniques, the most common, universal, and low tech strategy for
When the code is executed, you get a window into what the variables look like during executio

This is called **print debugging** and it is an incredibly useful tool.

::: callout-caution
#### Example: Nested Functions

::: panel-tabset
#### R
```

````
Imagine we start with this:



::: {.cell}

```{.r .cell-code}
x = 1
y = 2
z = 0

aa <- function(x) {
  bb <- function(y) {
    cc <- function(z) {
      z + y
    }
    cc(3) + 2
  }
  x + bb(4)
}

aa(5)
## [1] 14
```

and the goal is to understand what's happening in the code. We might add some lines:
````

```r
x = 1
y = 2
z = 0

aa <- function(x) {
  print(paste("Entering aa(). x = ", x))
  bb <- function(y) {
    print(paste("Entering bb(). x = ", x, "y = ", y))
    cc <- function(z) {
      print(paste("Entering cc(). x = ", x, "y = ", y, "z = ", z))
      cres <- z + y
      print(paste("Returning", cres, "from cc()"))
      cres
    }
    bres <- cc(3) + 2
    print(paste("Returning", bres, "from bb()"))
    bres
  }
  ares <- x + bb(4)
  print(paste("Returning",ares, "from aa()"))
  ares
}

aa(5)
## [1] "Entering aa(). x =  5"
## [1] "Entering bb(). x =  5 y =  4"
## [1] "Entering cc(). x =  5 y =  4 z =  3"
## [1] "Returning 7 from cc()"
## [1] "Returning 9 from bb()"
## [1] "Returning 14 from aa()"
## [1] 14
```

13.0.0.1 Python

Imagine we start with this:

```
x = 1
y = 2
z = 0

def aa(x):
  def bb(y):
    def cc(z):
      return z + y
    return cc(3) + 2
  return x + bb(4)

aa(5)
## 14
```

and the goal is to understand what's happening in the code. We might add some lines:

```
x = 1
y = 2
z = 0

def aa(x):
  print("Entering aa(). x = " + str(x))
  def bb(y):
    print("Entering bb(). x = " + str(x) + ", y = " + str(y))
    def cc(z):
      print("Entering cc(). x = " + str(x) + ", y = " + str(y) + ", z = " + str(z))
      cres = z + y
      print("Returning " + str(cres) + " from cc()")
      return cres
    bres = cc(3) + 2
    print("Returning " + str(bres) + " from bb()")
    return bres
  ares = x + bb(4)
  print("Returning " + str(ares) + " from aa()")
  return ares

aa(5)
## Entering aa(). x = 5
## Entering bb(). x = 5, y = 4
## Entering cc(). x = 5, y = 4, z = 3
## Returning 7 from cc()
## Returning 9 from bb()
## Returning 14 from aa()
## 14
```

:::

For more complex data structures, it can be useful to add `str()`, `head()`, or `summary()` functions.

Real world example: Web Scraping

In fall 2020, I wrote a webscraper to get election polling data from the RealClearPolitics site as part of the `electionViz` package. I wrote the function

`search_for_parent()` to get the parent HTML tag which matched the "tag" argument, that had the "node" argument as a descendant. I used print debugging to show the sequence of tags on the page.
I was assuming that the order of the parents would be "html", "body", "div", "table", "tbody", "tr" - descending from outer to inner (if you know anything about HTML/XML structure).
To prevent the site from changing on me (as websites tend to do…), I've saved the HTML file here.

13.0.0.1.1 R

```
library(xml2) # read html

search_for_parent <- function(node, tag) {
  # Get all of the parent nodes
  parents <- xml2::xml_parents(node)
  # Get the tags of every parent node
  tags <- purrr::map_chr(parents, rvest::html_name)
  print(tags)

  # Find matching tags
  matches <- which(tags == tag)
  print(matches)

  # Take the minimum matching tag
  min_match <- min(matches)
  if (length(matches) == 1) return(parents[min_match]) else return(NULL)
}

page <- read_html("shorturl.at/jkS59")
## Error: 'shorturl.at/jkS59' does not exist in current working directory ('/__w/stat-comp
# find all poll results in any table
poll_results <- xml_find_all(page, "//td[@class='lp-results']")
## Error in UseMethod("xml_find_all"): no applicable method for 'xml_find_all' applied to
# find the table that contains it
search_for_parent(poll_results[1], "table")
## Error in nodeset_apply(x, function(x) .Call(node_parents, x)): object 'poll_results' no
```

13.0.0.1.2 Python

You may need to `pip install lxml requests bs4` to run this code.

```
# !pip install lxml requests bs4
from bs4 import BeautifulSoup
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
import requests as req
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
import numpy as np


def search_for_parent(node, tag):
  # Get all of the parent nodes
  parents = node.find_parents()
  # get tag type for each parent node
  tags = [x.name for x in parents]
  print(tags)

  # Find matching tags
  matches = np.array([i for i, val in enumerate(tags) if val == tag])
  print(matches)

  # Take the minimum matching tag
  min_match = np.min(matches)
  if matches.size == 1:
    ret = parents[min_match]

  return ret


html_file = open('shorturl.at/jkS59', 'r')
## Error in py_call_impl(callable, dots$args, dots$keywords): FileNotFoundError: [Errno 2]
page = html_file.read()
# Read the page as HTML
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'html_file'
soup = BeautifulSoup(page, 'html')
# Find all poll results in any table
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'BeautifulSo
poll_results = soup.findAll('td', {'class': 'lp-results'})
# Find the table that contains the first poll result
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'soup' is no
search_for_parent(poll_results[0], 'table')
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'poll_result
```

By printing out all of the tags that contain `node`, I could see the order – inner to outer. I asked the function to return the location of the first table node, so the index (2nd value printed out) should match table in the character vector that was printed out first. I could then see that the HTML node that is returned is in fact the table node.

Try it out: Hurricanes in R

Not all bugs result in error messages, unfortunately, which makes higher-level techniques like `traceback()` less useful. The low-tech debugging tools, however, still work wonderfully.

13.0.0.1.3 Setup

```
library(ggplot2)
## Error in library(ggplot2): there is no package called 'ggplot2'
library(dplyr)
## Error in library(dplyr): there is no package called 'dplyr'
library(magrittr)
library(maps)
## Error in library(maps): there is no package called 'maps'
library(ggthemes)
## Error in library(ggthemes): there is no package called 'ggthemes'
worldmap <- map_data("world")
## Error in map_data("world"): could not find function "map_data"

# Load the data
data(storms, package = "dplyr")
## Error in find.package(package, lib.loc, verbose = verbose): there is no package called
```

13.0.0.1.4 Buggy code

The code below is supposed to print out a map of the tracks of all hurricanes of a specific category, 1 to 5, in 2013. Use print statements to figure out what's wrong with my code.

```
# Make base map to be used for each iteration
basemap <-  ggplot() +
  # Country shapes
  geom_polygon(aes(x = long, y = lat, group = group),
               data = worldmap, fill = "white", color = "black") +
  # Zoom in
  coord_quickmap(xlim = c(-100, -10), ylim = c(10, 50)) +
  # Don't need scales b/c maps provide their own geographic context...
  theme_map()
## Error in ggplot(): could not find function "ggplot"

for (i in 1:5) {
  # Subset the data
  subdata <- storms %>%
    filter(year == 2013) %>%
    filter(status == i)

  # Plot the data - path + points to show the observations
  plot <- basemap +
    geom_path(aes(x = long, y = lat, color = name), data = subdata) +
    geom_point(aes(x = long, y = lat, color = name), data = subdata) +
    ggtitle(paste0("Category ", i, " storms in 2013"))
  print(plot)
}
## Error in as.ts(x): object 'storms' not found
```

13.0.0.1.5 Solution 1: Identification

First, lets split the setup from the loop.

```
# Make base map to be used for each iteration
basemap <-  ggplot() +
  # Country shapes
  geom_polygon(aes(x = long, y = lat, group = group),
               data = worldmap, fill = "white", color = "black") +
  # Zoom in
  coord_quickmap(xlim = c(-100, -10), ylim = c(10, 50)) +
  # Don't need scales b/c maps provide their own geographic context...
  theme_map()
## Error in ggplot(): could not find function "ggplot"

print(basemap) # make sure the basemap is fine
## Error in print(basemap): object 'basemap' not found

# Load the data
data(storms, package = "dplyr")
## Error in find.package(package, lib.loc, verbose = verbose): there is no package called

str(storms) # make sure the data exists and is formatted as expected
## Error in str(storms): object 'storms' not found
```

Everything looks ok in the setup chunk…

```
for (i in 1:5) {
  print(paste0("Category ", i, " storms"))
  # Subset the data
  subdata <- storms %>%
    filter(year == 2013) %>%
    filter(status == i)

  print(paste0("subdata dims: nrow ", nrow(subdata), " ncol ", ncol(subdata)))
        # str(subdata) works too, but produces more clutter. I started
        # with str() and moved to dim() when I saw the problem

  # Plot the data - path + points to show the observations
  plot <- basemap +
    geom_path(aes(x = long, y = lat, color = name), data = subdata) +
    geom_point(aes(x = long, y = lat, color = name), data = subdata) +
    ggtitle(paste0("Category ", i, " storms in 2013"))
  # print(plot) # Don't print plots - clutters up output at the moment
}
## [1] "Category 1 storms"
## Error in as.ts(x): object 'storms' not found
```

Ok, so from this we can see that something is going wrong with our filter statement - we have no rows of data.

13.0.0.1.6 Solution 2: Fixing

```
head(storms)
## Error in head(storms): object 'storms' not found
```

Whoops. I meant “category” when I typed “status”.

```
for (i in 1:5) {
  print(paste0("Category ", i, " storms"))
  # Subset the data
  subdata <- storms %>%
    filter(year == 2013) %>%
    filter(category == i)

  print(paste0("subdata dims: nrow ", nrow(subdata), " ncol ", ncol(subdata)))
        # str(subdata) works too, but produces more clutter. I started
        # with str() and moved to dim() when I saw the problem

  # Plot the data - path + points to show the observations
  plot <- basemap +
    geom_path(aes(x = long, y = lat, color = name), data = subdata) +
    geom_point(aes(x = long, y = lat, color = name), data = subdata) +
    ggtitle(paste0("Category ", i, " storms in 2013"))
  # print(plot) # Don't print plots - clutters up output at the moment
}
## [1] "Category 1 storms"
## Error in as.ts(x): object 'storms' not found
```

Ok, that's something, at least. We now have some data for category 1 storms…

```
filter(storms, year == 2013) %>%
  # Get max category for each named storm
  group_by(name) %>%
  filter(category == max(category)) %>%
  ungroup() %>%
  # See what categories exist
  select(name, category) %>%
  unique()
## Error in select(., name, category): could not find function "select"
```

It looks like 2013 was just an incredibly quiet year for tropical activity.

13.0.0.1.7 Solution 3: Verifying

2013 may have been a quiet year for tropical activity in

the Atlantic, but 2004 was not. So let's just make sure our code works by checking out 2004.

```
for (i in 1:5) {
  print(paste0("Category ", i, " storms"))
  # Subset the data
  subdata <- storms %>%
    filter(year == 2004) %>%
    filter(category == i)

  print(paste0("subdata dims: nrow ", nrow(subdata), " ncol ", ncol(subdata)))
        # str(subdata) works too, but produces more clutter. I started
        # with str() and moved to dim() when I saw the problem

  # Plot the data - path + points to show the observations
  plot <- basemap +
    geom_path(aes(x = long, y = lat, color = name), data = subdata) +
    geom_point(aes(x = long, y = lat, color = name), data = subdata) +
    ggtitle(paste0("Category ", i, " storms in 2013"))
  print(plot) # Don't print plots - clutters up output at the moment
}
## [1] "Category 1 storms"
## Error in as.ts(x): object 'storms' not found
```

If we want to only print informative plots, we could add an if statement. Now that the code works, we can also comment out our print() statements (we could delete them, too, depending on whether we anticipate future problems with the code).

```r
for (i in 1:5) {
  # print(paste0("Category ", i, " storms"))

  # Subset the data
  subdata <- storms %>%
    filter(year == 2013) %>%
    filter(category == i)

  # print(paste0("subdata dims: nrow ", nrow(subdata), " ncol ", ncol(subdata)))
  #       # str(subdata) works too, but produces more clutter. I started
  #       # with str() and moved to dim() when I saw the problem

  # Plot the data - path + points to show the observations
  plot <- basemap +
    geom_path(aes(x = long, y = lat, color = name), data = subdata) +
    geom_point(aes(x = long, y = lat, color = name), data = subdata) +
    ggtitle(paste0("Category ", i, " storms in 2013"))

  if (nrow(subdata) > 0) print(plot)
}
## Error in as.ts(x): object 'storms' not found
```

Once you've found your problem, go back and delete or comment out your print statements, as they're no longer necessary. If you think you may need them again, comment them out, otherwise, just delete them so that your code is neat, clean, and concise.

13.0.1 After an error has occurred - `traceback()`

`traceback()` can help you narrow down where an error occurs by taking you through the series of function calls that led up to the error. This may help you identify which function is actually causing the problem, which is especially useful when you have nested functions or are using package functions that depend on other packages.

Demo: Using traceback

13.0.1.0.1 R

```
aa <- function(x) {
  bb <- function(y) {
    cc <- function(z) {
     stop('there was a problem')  # This generates an error
    }
    cc()
  }
  bb()
}

aa()
## Error in cc(): there was a problem
```

For more information, you could run traceback

```
traceback()
```

Which will provide the following output:

```
4: stop("there was a problem") at #4
3: c() at #6
2: b() at #8
1: a()
```

Reading through this, we see that a() was called, b() was called, c() was called, and then there was an error. It's even kind enough to tell us that the error occurred at line 4 of the code.
If you are running this code interactively in RStudio, it's even easier to run `traceback()` by clicking on the "Show Traceback" option that appears when there is an error.

Figure 13.1: Both Show Traceback and Rerun with Debug are useful tools

If you are using `source()` to run the code in Rstudio, it will even provide a link to the file and line location of the error.

13.0.1.0.2 Python

```
import sys,traceback

def aa(x):
  def bb(y):
    def cc(z):
      try:
        return y + z + tuple()[0] # This generates an error
      except IndexError:
        exc_type, exc_value, exc_tb = sys.exc_info()
        traceback.print_exception(exc_type, exc_value, exc_tb, file = sys.stdout)
    return cc(3) + 2
  return x + bb(4)

aa(5)
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupported opera
```

Python's traceback information is a bit more low-level and requires a bit more from the programmer than R's version.

13.0.2 Interactive Debugging

13.0.2.1 R `browser()`

The `browser()` function is useful for debugging your own code. If you're writing a function and something isn't working quite right, you can insert a call to `browser()` in that function, and examine what's going on.

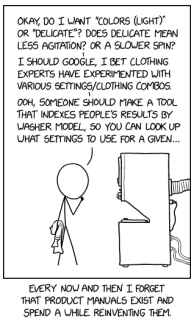
Example of using browser()

Suppose that I want to write a function that will plot an xkcd comic in R.
I start with

```
library(png)
library(xml2)
library(dplyr)
## Error in library(dplyr): there is no package called 'dplyr'

# get the most current xkcd
get_xkcd <- function() {
  url <- "http://xkcd.com"
  page <- read_html(url)
  # Find the comic
  image <- xml_find_first(page, "//div[@id='comic']/img") %>%
    # pull the address out of the tag
    xml_attr("src")


  readPNG(source = image)
}

get_xkcd() %>%
  as.raster() %>%
  plot()
## Error in readPNG(source = image): unable to open //imgs.xkcd.com/comics/washing_machine
```

Here's the final function

```
library(png)
library(xml2)

# get the most current xkcd
get_xkcd <- function() {

  url <- "http://xkcd.com"
  page <- read_html(url)
  # Find the comic
  image <- xml_find_first(page, "//div[@id='comic']/img") %>%
    # pull the address out of the tag
    xml_attr("src")

  # Fix image address so that we can access the image
  image <- substr(image, 3, nchar(image))

  # Download the file to a temp file and read from there
  file_location <- tempfile(fileext = ".png")
  download.file(image, destfile = file_location, quiet = T)

  readPNG(source = file_location)
}

get_xkcd() %>%
  as.raster() %>%
  plot()
```



13.0.2.2 Python

In python, the equivalent interactive debugger is `ipdb`. You can install it with `pip install ipdb`.

If you want to run Python in the interactive ipython console, then you can invoke the ipdb debugging with `%debug get_xkcd()`. This is similar to `browser()` in R. If you're working in Python in RStudio, though, you have to get into debug mode in a more involved way.

To run code using `ipdb` when your code hits an error, add `from ipdb import launch_ipdb_on_exception` to the top of your python code chunk. Then, at the bottom, put any lines that may trigger the error after these two lines:

```
if __name__ == "__main__":
  with launch_ipdb_on_exception():
    <your properly indented code goes here>
```

This ensures that ipdb is launched when an error is reached.

> Example using `ipdb`
>
> Suppose that I want to write a function that will plot an xkcd comic in python.
> I start with

```
from bs4 import BeautifulSoup
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
import urllib.request # work with html
from PIL import Image # work with images
import numpy as np

# importing pyplot and image from matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as img

# get the most current xkcd
def get_xkcd():
  url = "http://xkcd.com"

  # Get the URL
  html_file = urllib.request.urlopen(url)
  page = html_file.read()
  decode_page = page.decode("utf8")

  # Read the page as HTML
  soup = BeautifulSoup(decode_page, 'html')

  # Get the comic src from the img tag
  imlink = soup.select('#comic > img')[0].get('src')
  # Format as a numpy array
  image = np.array(Image.open(urllib.request.urlopen(imlink)))

  return image

plt.imshow(get_xkcd())
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'BeautifulSo
plt.show()
```

Here's the final function

```
from bs4 import BeautifulSoup
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
import urllib.request # work with html
from PIL import Image # work with images
import numpy as np

# importing pyplot and image from matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as img

# get the most current xkcd
def get_xkcd():
  url = "http://xkcd.com"

  # Get the URL
  html_file = urllib.request.urlopen(url)
  page = html_file.read()
  decode_page = page.decode("utf8")

  # Read the page as HTML
  soup = BeautifulSoup(decode_page, 'html')

  # Get the comic src from the img tag
  imlink = soup.select('#comic > img')[0].get('src')
  # Format as a numpy array
  image = np.array(Image.open(urllib.request.urlopen('https:' + imlink)))

  return image

plt.imshow(get_xkcd())
## Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'BeautifulSo
plt.show()
```

Try it out

13.0.2.2.1 Problem

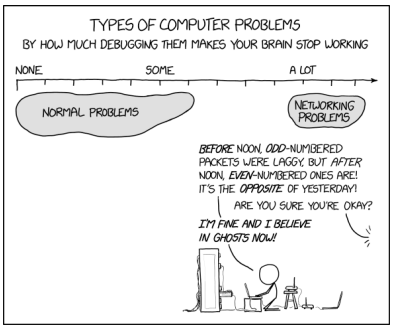
Each xkcd has a corresponding ID number (ordered sequentially from 1 to 2722 at the time this was written). Modify the XKCD functions above to make use of the id parameter, so that you can pass in an ID number and get the relevant comic.

Use interactive debugging tools to help you figure out what logic you need to add. You should not need to change the web scraping code - the only change should be to the URL.

What things might you add to make this function "defensive programming" compatible?

13.0.2.2.2 R Solution

```r
# get the most current xkcd or the specified number
get_xkcd <- function(id = NULL) {
  if (is.null(id)) {
    # Have to get the location of the image ourselves
    url <- "http://xkcd.com"
  } else if (is.numeric(id)) {
    url <- paste0("http://xkcd.com/", id, "/")
  } else {
    # only allow numeric or null input
    stop("To get current xkcd, pass in NULL, otherwise, pass in a valid comic number")
  }

  page <- read_html(url)
  # Find the comic
  image <- xml_find_first(page, "//div[@id='comic']/img") %>%
    # pull the address out of the tag
    xml_attr("src")
  # Fix image address so that we can access the image
  image <- substr(image, 3, nchar(image)) # cut the first 2 characters off

  # make temp file
  location <- tempfile(fileext = "png")
  download.file(image, destfile = location, quiet = T)

  # This checks to make sure we saved the file correctly
  if (file.exists(location)) {
    readPNG(source = location)
  } else {
    # Give a good informative error message
    stop(paste("Something went wrong saving the image at ", image, " to ", location))
  }
}

get_xkcd(2259) %>%
  as.raster() %>%
  plot()
```



13.0.2.2.3 Python Solution

```
from bs4 import BeautifulSoup
## Error in py_call_impl(callable, dots$args, dots$keywords): ModuleNotFoundError: No modu
import urllib.request # work with html
from PIL import Image # work with images
import numpy as np

# importing pyplot and image from matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as img

# get the most current xkcd
def get_xkcd(id=''):
  image = 0 # Defining a placeholder

  if id == '':
    # Have to get the location of the image ourselves
    url = "http://xkcd.com"
  elif id.isnumeric():
    url = "http://xkcd.com/" + id + "/"
  else:
    # only allow numeric or null input
    raise TypeError("To get current xkcd, pass in an empty string, otherwise, pass in a va

  # Print debugging left in for your amusement
  # print(type(id))

  # Get the URL
  html_file = urllib.request.urlopen(url)
  page = html_file.read()
  decode_page = page.decode("utf8")

  # Read the page as HTML
  soup = BeautifulSoup(decode_page, 'html')

  # Get the comic src from the img tag
  imnode = soup.select('#comic > img')

  try:
    imlink = imnode[0].get('src')
  except:
    raise Exception("No comic could be found with number " + id + " (url = "+ url+ " )")

  try:
    # Format as a numpy array
    image = np.array(Image.open(urllib.request.urlopen('https:' + imlink)))
    return image
  except:
    raise Exception("Reading the image failed. Check to make sure an image exists at " + u
    return(None)
```

```
res = get_xkcd('')
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'BeautifulSoup' is
```

```
plt.imshow(res)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): NameError: name 'res' is not defin
```

```
plt.show()
```

```
res = get_xkcd('3000')
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): urllib.error.HTTPError: HTTP Error
```

```
res = get_xkcd('abcd')
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: To get current xkcd, pa
```

13.0.3 R `debug()`

In the `traceback()` Rstudio output, the other option is "rerun with debug". In short, debug mode opens up a new interactive session inside the function evaluation environment. This lets you observe what's going on in the function, pinpoint the error (and what causes it), and potentially fix the error, all in one neat workflow.

`debug()` is most useful when you're working with code that you didn't write yourself. So, if you can't change the code in the function causing the error, `debug()` is the way to go. Otherwise, using `browser()` is generally easier. Essentially, `debug()` places a `browser()` statement at the first line of a function, but without having to actually alter the function's source code.

`debug()` example

```
data(iris)

tmp <- lm(Species ~ ., data = iris)
summary(tmp)
##
## Call:
## lm(formula = Species ~ ., data = iris)
##
## Residuals:
## Error in quantile.default(resid): (unordered) factors are not allowed
```

We get this weird warning, and then an error about factors when we use summary() to look at the coefficients.

```
debug(lm) # turn debugging on


tmp <- lm(Species ~ ., data = iris)
summary(tmp)

undebug(lm) # turn debugging off
```

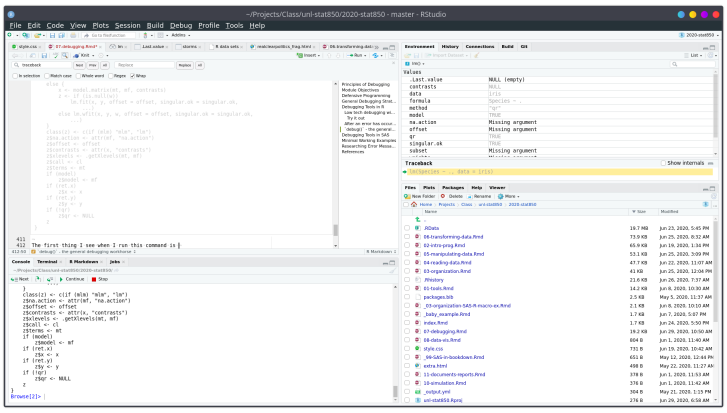


Figure 13.2: The first thing I see when I run lm after turning on debug (screenshot)

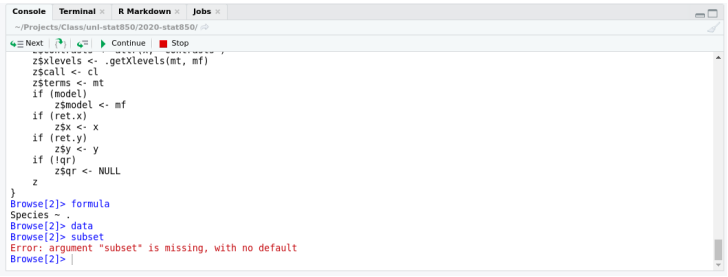


Figure 13.3: The variables passed into the lm function are available as named and used in the function. In addition, we have some handy buttons in the console window that will let us 'drive' through the function

After pressing "next" a few times, you can see that I've stepped through the first few lines of the lm function.

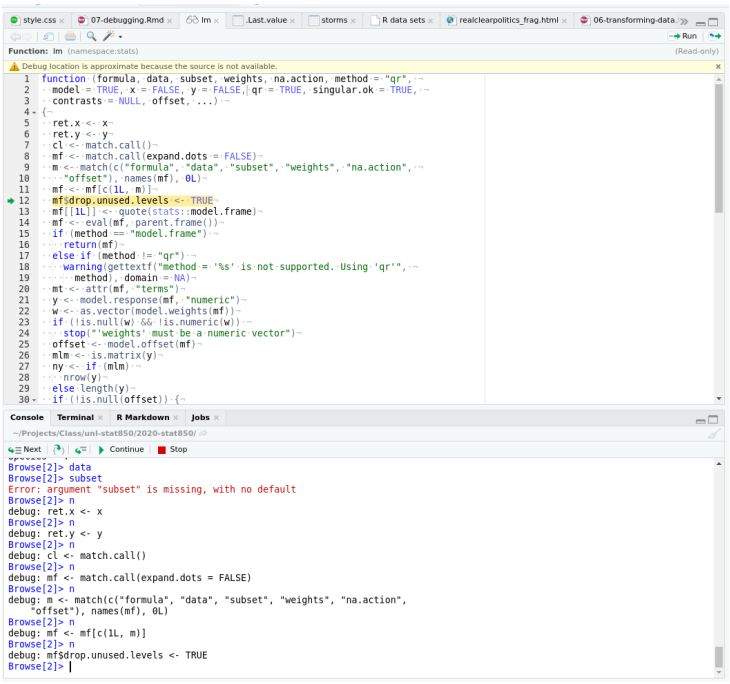


Figure 13.4: Stepping through the function. The arrow on the left side in the editor window shows which line of code we're currently at.

We can see that once we're at line 21, we get a warning about using type with a factor response, and that the warning occurs during a call to the `model.response` function. So, we've narrowed our problem down - we passed in a numeric variable as the response (y) variable, but it's a factor, so our results aren't going to mean much. We were using the function wrong.

We probably could have gotten there from reading the error message carefully, but this has allowed us to figure out exactly what happened, where it happened, and why it happened.

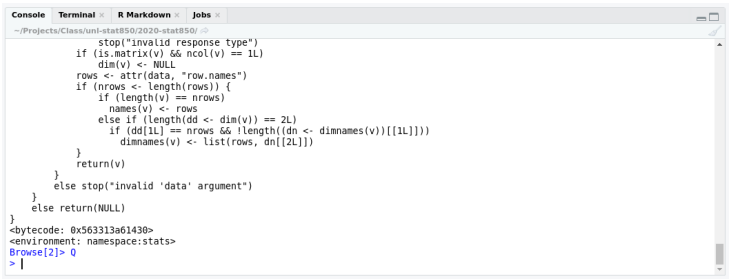


Figure 13.5: I can hit "Stop" or type "Q" to exit the debug environment.

But, until I run `undebug(lm)`, every call to `lm` will take me into the debug window.

`undebug(f)` will remove the debug flag on the function `f`. `debugonce(f)` will only debug f the first time it is run.

Try it out: `debug` in R

13.0.3.0.1 Problem

`larger(x, y)` is supposed to return the elementwise maximum of two vectors.

```
larger <- function(x, y) {
  y.is.bigger <- y > x
  x[y.is.bigger] <- y[y.is.bigger]
  x
}

larger(c(1, 5, 10), c(2, 4, 11))
## [1]  2  5 11
```

```
larger(c(1, 5, 10), 6)
## [1]  6 NA 10
```

Why is there an NA in the second example? It should be a 6. Figure out why this happens, then try to fix it.

13.0.3.0.2 Solution

I'll replicate "debug" in non-interactive mode by setting up an environment where x and y are defined

```
x <- c(1, 5, 10)
y <- 6

# Inside of larger() with x = c(1, 5, 10), y = 6
(y.is.bigger <- y > x ) # putting something in () prints it out
## [1]  TRUE  TRUE FALSE
y[y.is.bigger] # This isn't quite what we were going for, but it's what's causing the issu
## [1]  6 NA
x[y.is.bigger] # What gets replaced
## [1] 1 5


# Better option
larger <- function(x, y) {
  y.is.bigger <- y > x
  ifelse(y.is.bigger, y, x)
}
```

14 Matrix Calculations

While R, SAS, and Python are all extremely powerful statistical programming languages, the core of most programming languages is the ability to do basic calculations and matrix arithmetic. As almost every dataset is stored as a matrix-like structure (data sets and data frames both allow for multiple types, which isn't quite compatible with more canonical matrices), it is useful to know how to do matrix-level calculations in whatever language you are planning to use to work with data.

In this section, we will essentially be using our programming language as overgrown calculators.

In the next chapters we'll talk about data types and structures, so you'll get to see more about matrices and arrays, but for now, let's confine ourselves to using R and python to do basic math calculations.

Table 14.1: Table of common mathematical and matrix operations in R, SAS, and Python [32].

Operation	R	SAS	Python
Addition	+	+	+
Subtraction	-	-	-
Elementwise Multiplication	*	#	*
Division	/	/	/
Modulo (Remainder)	%%	MOD	%
Integer Division	%/%	FLOOR(x\y)	//
Elementwise Exponentiation	^	##	**
Matrix/Vector Multiplication	%*%	*	`np.dot()`

Operation	R	SAS	Python
Matrix Exponentiation	`^`	**	`np.exp()`
Matrix Transpose	`t(A)`	`A'`	`np.transpose(A)`
Matrix Determinant	`det(A)`	`det(A)`	`np.linalg.det(A)`
Matrix Diagonal	`diag(A)`	`diag(A)`	`np.linalg.diag(A)`
Matrix Inverse	`solve(A)`	`solve(A, diag({...}))`	`np.linalg.inv(A)`

14.0.1 Basic Mathematical Operators

R

```
x <- 1:10
y <- seq(3, 30, by = 3)

x + y
##  [1]  4  8 12 16 20 24 28 32 36 40
x - y
##  [1]  -2  -4  -6  -8 -10 -12 -14 -16 -18 -20
x * y
##  [1]   3  12  27  48  75 108 147 192 243 300
x / y
##  [1] 0.3333333 0.3333333 0.3333333 0.3333333 0.3333333 0.3333333 0.3333333
##  [8] 0.3333333 0.3333333 0.3333333
x^2
##  [1]   1   4   9  16  25  36  49  64  81 100
t(x) %*% y
##      [,1]
## [1,] 1155
```

Python

```
import numpy as np

x = np.array(range(1, 11))
y = np.array(range(3, 33, 3)) # python indexes are not inclusive

x + y
## array([ 4,  8, 12, 16, 20, 24, 28, 32, 36, 40])
x - y
## array([ -2,  -4,  -6,  -8, -10, -12, -14, -16, -18, -20])
x * y
## array([  3,  12,  27,  48,  75, 108, 147, 192, 243, 300])
x / y
## array([0.33333333, 0.33333333, 0.33333333, 0.33333333, 0.33333333,
##        0.33333333, 0.33333333, 0.33333333, 0.33333333, 0.33333333])
x ** 2
## array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
np.dot(x.T, y)
## 1155
```

SAS

By default, SAS creates row vectors with `do(a, b, by = c)` syntax. The transpose operator (a single backtick) can be used to transform `A` into `A`‘.

```
proc iml;
  x = do(1, 10, 1);
  y = do(3, 30, 3);

  z = x + y;
  z2 = x - y;
  z3 = x # y;
  z4 = x/y;
  z5 = x##2;
  z6 = x` * y;
  print z, z2, z3, z4, z5, z6;
quit;
```

14.0.2 Matrix Operations

Other matrix operations, such as determinants and extraction of the matrix diagonal, are similarly easy:

R

```
mat <- matrix(c(1, 2, 3, 6, 4, 5, 7, 8, 9), nrow = 3, byrow = T)
mat
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    6    4    5
## [3,]    7    8    9
t(mat) # transpose
##      [,1] [,2] [,3]
## [1,]    1    6    7
## [2,]    2    4    8
## [3,]    3    5    9
det(mat) # get the determinant
## [1] 18
diag(mat) # get the diagonal
## [1] 1 4 9
diag(diag(mat)) # get a square matrix with off-diag 0s
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    4    0
## [3,]    0    0    9
diag(1:3) # diag() also will create a diagonal matrix if given a vector
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

Python

```
import numpy as np
mat = np.array([[1, 2, 3],[6, 4, 5],[7, 8, 9]], dtype = int, order ='C')

mat
## array([[1, 2, 3],
##        [6, 4, 5],
##        [7, 8, 9]])
mat.T
## array([[1, 6, 7],
##        [2, 4, 8],
##        [3, 5, 9]])
np.linalg.det(mat) # numerical precision...
## 18.000000000000004
np.diag(mat)
## array([1, 4, 9])
np.diag(np.diag(mat))
## array([[1, 0, 0],
##        [0, 4, 0],
##        [0, 0, 9]])
np.diag(range(1, 4))
## array([[1, 0, 0],
##        [0, 2, 0],
##        [0, 0, 3]])
```

SAS

```
proc iml;
  mat = {1 2 3, 6 4 5, 7 8 9};
  tmat = mat`; /* transpose */
  determinant = det(mat); /* get the determinant */
  diagonal_vector = vecdiag(mat); /* get the diagonal as a vector */
  diagonal_mat = diag(mat); /* get the diagonal as a square matrix */
                            /* with 0 on off-diagonal entries */

  dm = diag({1 2 3}); /* make a square matrix with vector as the diagonal */

  print tmat, determinant, diagonal_vector, diagonal_mat, dm;
```

```
quit;
```

14.0.3 Matrix Inverse

The other important matrix-related function is the inverse. In R, `A^-1` will get you the elementwise reciprocal of the matrix. Not exactly what we'd like to see… Instead, in R and SAS, we use the `solve()` function. The inverse is defined as the matrix B such that `AB = I` where `I` is the identity matrix (1's on diagonal, 0's off-diagonal). So if we `solve(A)` (in R) or `solve(A, diag(n))` in SAS (where n is a vector of 1s the size of A), we will get the inverse matrix. In Python, we use the `np.linalg.inv()` function to invert a matrix, which is a bit more linguistically familiar.

R

```
mat <- matrix(c(1, 2, 3, 6, 4, 5, 7, 8, 9), nrow = 3, byrow = T)

minv <- solve(mat) # get the inverse

minv
##            [,1]       [,2]       [,3]
## [1,] -0.2222222  0.3333333 -0.1111111
## [2,] -1.0555556 -0.6666667  0.7222222
## [3,]  1.1111111  0.3333333 -0.4444444
mat %*% minv
##                [,1]         [,2]          [,3]
## [1,]  1.000000e+00 0.000000e+00  1.110223e-16
## [2,] -8.881784e-16 1.000000e+00 -5.551115e-16
## [3,]  0.000000e+00 2.220446e-16  1.000000e+00
```

Python

```
import numpy as np
mat = np.array([[1, 2, 3],[6, 4, 5],[7, 8, 9]], dtype = int, order ='C')

minv = np.linalg.inv(mat)
minv
## array([[-0.22222222,  0.33333333, -0.11111111],
##        [-1.05555556, -0.66666667,  0.72222222],
##        [ 1.11111111,  0.33333333, -0.44444444]])
np.dot(mat, minv)
## array([[ 1.00000000e+00,  0.00000000e+00,  1.11022302e-16],
##        [-8.88178420e-16,  1.00000000e+00, -5.55111512e-16],
##        [ 0.00000000e+00,  2.22044605e-16,  1.00000000e+00]])
np.round(np.dot(mat, minv), 2)
## array([[ 1.,  0.,  0.],
##        [-0.,  1., -0.],
##        [ 0.,  0.,  1.]])
```

SAS

Documentation

```
proc iml;
  mat = {1 2 3, 6 4 5, 7 8 9};

  mat_inv = solve(mat, diag({1 1 1})); /* get the inverse */
  mat_inv2 = inv(mat); /* less efficient and less accurate */
  print mat_inv, mat_inv2;

  id = mat * mat_inv;
  id2 = mat * mat_inv2;
  print id, id2;
quit;
```

Part III

Part III: Data Wrangling

This part of the textbook covers topics related to working with data. Every data set is messy in its own way [33], and this section is focused on providing you with some of the tools to deal with the most common types of messy data.

Chapter 15 covers how to read in data from many common formats, such as spreadsheets, web tables, and databases.

Chapter 16 provides a brief primer on how to create different charts and graphics in R and python for use in other sections.

Chapter 17 covers verbs for transforming data - creating new variables, modifying existing variables, selecting specific rows and/or columns, and more.

Chapter 18 discusses string manipulations. Text data is some of the messiest data out there, and this chapter will give you some tools to help make textual data more tidy.

Chapter 19 discusses how to transform data from wide human-friendly formats to long comptuer-friendly formats for analysis and processing.

Chapter 20 discusses how to join data sets together using common variables.

Chapter 21 discusses how to work with dates and times using R and Python.

Chapter 22 discusses how to work with data stored in nested lists efficiently.

Chapter 23 covers working with spatial data formats: drawing maps, working with spatial regions, and more. We will focus exclusively on the visualization and data wrangling part, leaving the modeling of spatial data for a different time.

References

15 Data Input

16 Data Visualization Basics

17 Data Cleaning

18 Working with Strings

19 Reshaping Data

20 Joining Data

21 Dates and Times

22 List data structures

23 Spatial data

24 Other Topics

24.1 Mathematical Logic

In Chapter 8 and Chapter 9 we talk about more complicated data structures and control structures (for loops, if statements). I've included this section because it may be useful to review some concepts from mathematical logic.

Unfortunately, to best demonstrate mathematical logic, I'm going to need you to know that a vector is like a list of the same type of thing. In R, vectors are defined using `c()`, so `c(1, 2, 3)` produces a vector with entries 1, 2, 3. In Python, we'll primarily use `numpy` arrays, which we create using `np.array([1, 2, 3])`. Technically, this is creating a list, and then converting that list to a numpy array.

24.1.1 And, Or, and Not

We can combine logical statements using and, or, and not.

- (X AND Y) requires that both X and Y are true.
- (X OR Y) requires that one of X or Y is true.
- (NOT X) is true if X is false, and false if X is true. Sometimes called **negation**.

In R, we use `!` to symbolize NOT, in Python, we use `~` for vector-wise negation (NOT).

Order of operations dictates that NOT is applied before other operations. So `NOT X AND Y` is read as `(NOT X) AND (Y)`. You must use parentheses to change the way this is interpreted.

24.1.1.1 R

```
x <- c(TRUE, FALSE, TRUE, FALSE)
y <- c(TRUE, TRUE, FALSE, FALSE)

x & y # AND
## [1]  TRUE FALSE FALSE FALSE
x | y # OR
## [1]  TRUE  TRUE  TRUE FALSE
!x & y # NOT X AND Y
## [1] FALSE  TRUE FALSE FALSE
x & !y # X AND NOT Y
## [1] FALSE FALSE  TRUE FALSE
```

24.1.1.2 Python

```
import numpy as np
x = np.array([True, False, True, False])
y = np.array([True, True, False, False])

x & y
## array([ True, False, False, False])
x | y
## array([ True,  True,  True, False])
~x & y
## array([False,  True, False, False])
x & ~y
## array([False, False,  True, False])
```

24.1.2 De Morgan's Laws

De Morgan's Laws are a set of rules for how to combine logical statements. You can represent them in a number of ways:

- NOT(A or B) is equivalent to NOT(A) and NOT(B)
- NOT(A and B) is equivalent to NOT(A) or NOT(B)

We can also represent them with Venn Diagrams.

24.1.2.1 Definitions

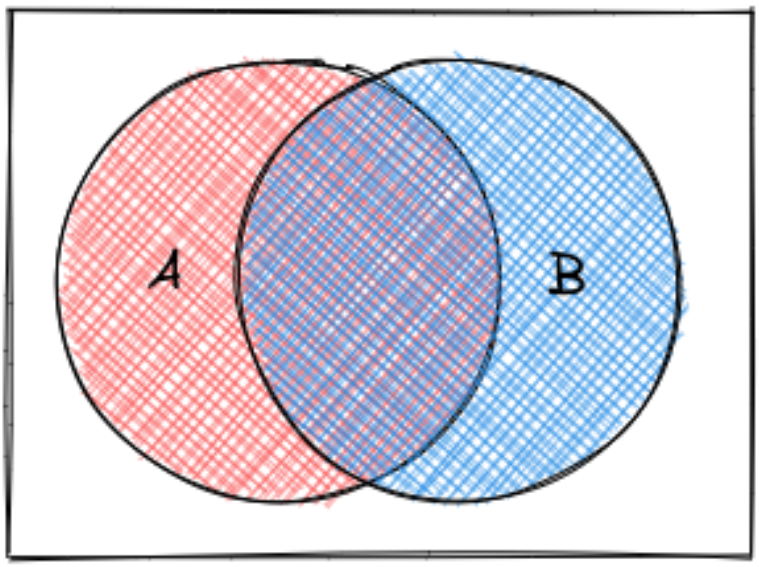
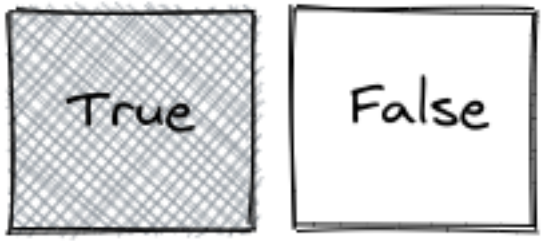


Figure 24.1: Venn Diagram of Set A and Set B

Suppose that we set the convention that 

.

24.1.2.2 DeMorgan's First Law

24.1.2.3 DeMorgan's Second Law

24.2 Controlling Loops with Break, Next, Continue

Sometimes it is useful to control the statements in a loop with a bit more precision. You may want to skip over code and

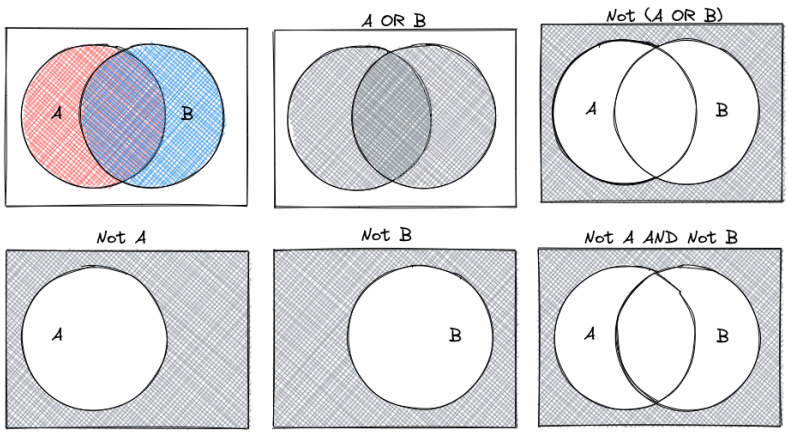


Figure 24.2: A venn diagram illustration of De Morgan's laws showing that the region that is outside of the union of A OR B (aka NOT (A OR B)) is the same as the region that is outside of (NOT A) and (NOT B)

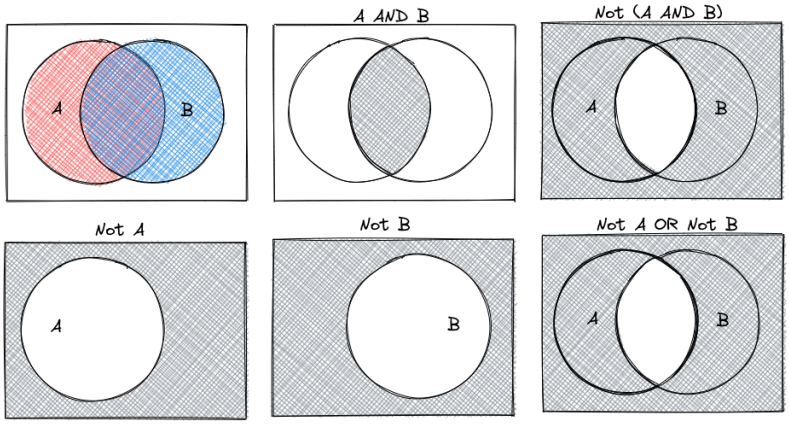


Figure 24.3: A venn diagram illustration of De Morgan's laws showing that the region that is outside of the union of A AND B (aka NOT (A AND B)) is the same as the region that is outside of (NOT A) OR (NOT B)

proceed directly to the next iteration, or, as demonstrated in the previous section with the `break` statement, it may be useful to exit the loop prematurely.

24.2.1 Break Statement

24.2.2 Next/Continue Statement

> ⚠ Example: Next/continue and Break statements
>
> Let's demonstrate the details of next/continue and break statements.
> We can do different things based on whether i is evenly divisible by 3, 5, or both 3 and 5 (thus divisible by 15)

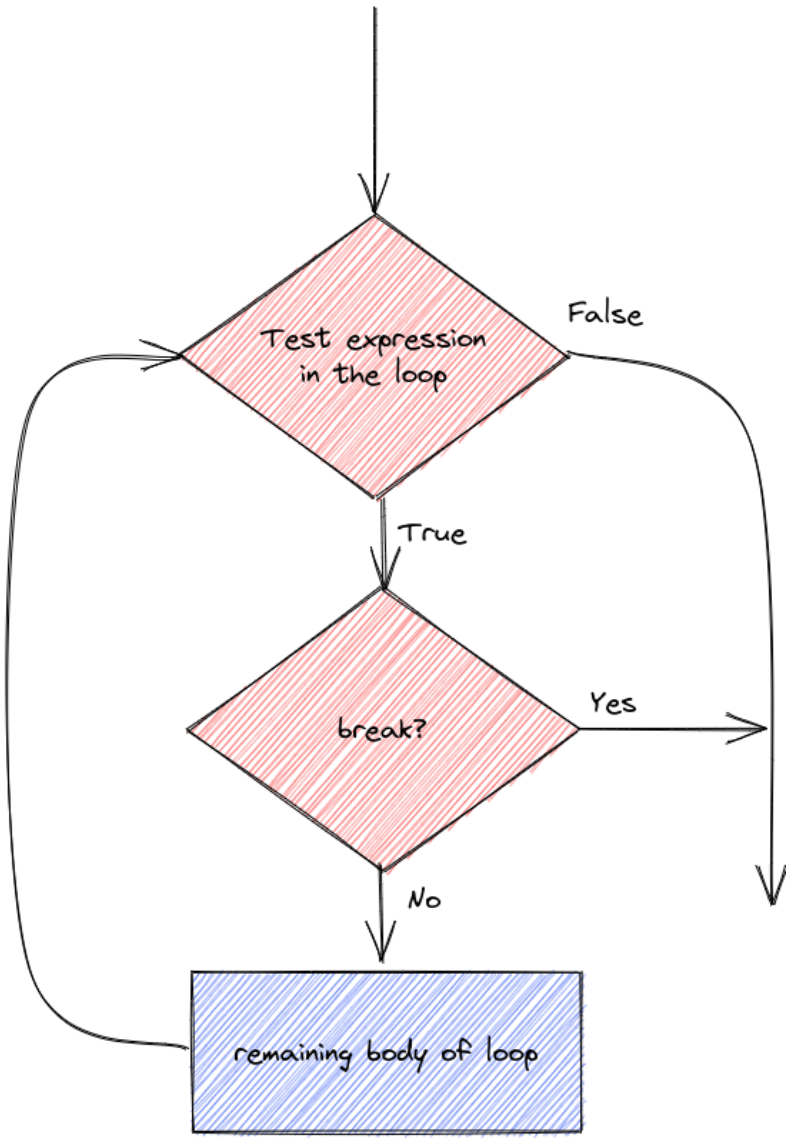


Figure 24.4: A break statement is used to exit a loop prematurely

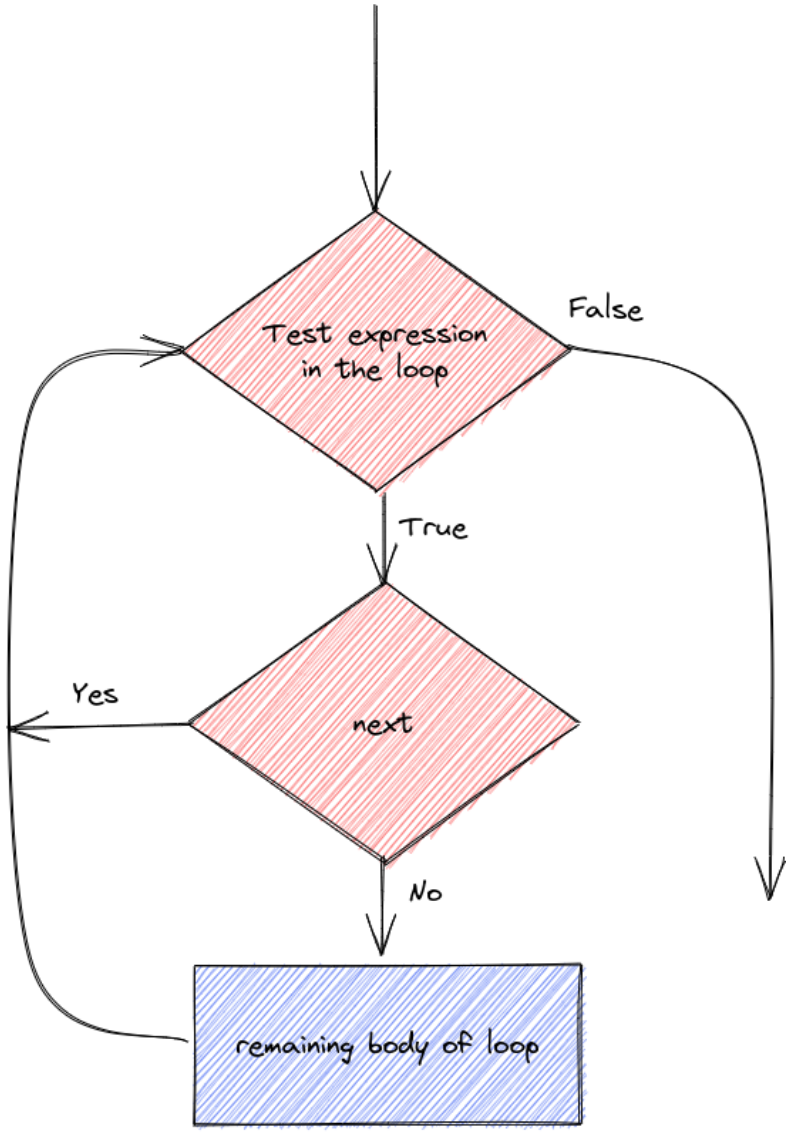


Figure 24.5: A next (or continue) statement is used to skip the body of the loop and continue to the next iteration

R

```r
for (i in 1:20) {
  if (i %% 15 == 0) {
    print("Exiting now")
    break
  } else if (i %% 3 == 0) {
    print("Divisible by 3")
    next
    print("After the next statement") # this should never execute
  } else if (i %% 5 == 0) {
    print("Divisible by 5")
  } else {
    print(i)
  }
}
## [1] 1
## [1] 2
## [1] "Divisible by 3"
## [1] 4
## [1] "Divisible by 5"
## [1] "Divisible by 3"
## [1] 7
## [1] 8
## [1] "Divisible by 3"
## [1] "Divisible by 5"
## [1] 11
## [1] "Divisible by 3"
## [1] 13
## [1] 14
## [1] "Exiting now"
```

Python

```
for i in range(1, 20):
  if i%15 == 0:
    print("Exiting now")
    break
  elif i%3 == 0:
    print("Divisible by 3")
    continue
    print("After the next statement") # this should never execute
  elif i%5 == 0:
    print("Divisible by 5")
  else:
    print(i)
## 1
## 2
## Divisible by 3
## 4
## Divisible by 5
## Divisible by 3
## 7
## 8
## Divisible by 3
## Divisible by 5
## 11
## Divisible by 3
## 13
## 14
## Exiting now
```

To be quite honest, I haven't really ever needed to use next/continue statements when I'm programming, and I rarely use break statements. However, it's useful to know they exist just in case you come across a problem where you could put either one to use.

24.3 Recursion

Under construction.

In the meantime, check out [34] (R) and [35] (Python) for decent coverage of the basic idea of recursive functions.

References

[1] D. Robitzski, "Gen z kids apparently don't understand how file systems work. Futurism," Sep. 24, 2021. [Online]. Available: https://futurism.com/the-byte/gen-z-kids-file-systems. [Accessed: Jan. 09, 2023]

[2] "Punched card input/output." Jan. 08, 2023 [Online]. Available: https://en.wikipedia.org/w/index.php?title=Punched_card_input/output&oldid=1132250858

[3] Posit PBC, "Quarto - markdown basics," 2023. [Online]. Available: https://quarto.org/docs/authoring/markdown-basics.html. [Accessed: Jan. 09, 2023]

[4] Y. Xie, "The first notebook war," Sep. 10, 2018. [Online]. Available: https://yihui.org/en/2018/09/notebook-war/. [Accessed: Jan. 09, 2023]

[5] J. Bryan, J. Hester, and {The Stat 545 TAs}, *Happy git and GitHub for the useR*. 2021 [Online]. Available: https://happygitwithr.com/. [Accessed: May 09, 2022]

[6] dustbuster, "Answer to "git is not working after macOS update (xcrun: Error: Invalid active developer path (/library/developer/CommandLineTools)". Stack overflow," Sep. 26, 2018. [Online]. Available: https://stackoverflow.com/a/52522566/2859168. [Accessed: Jan. 13, 2023]

[7] D. C. R. Severance, *Python for Everybody: Exploring Data in Python 3*. Ann Arbor, MI: CreateSpace Independent Publishing Platform, 2016 [Online]. Available: https://www.py4e.com/html3/

[8] R. Ihaka, "R : Past and future history," 1998 [Online]. Available: https://www.stat.auckland.ac.nz/~ihaka/downloads/Interface98.pdf

[9] *Why TRUE + TRUE = 2: Data Types*. (Feb. 03, 2020) [Online]. Available: https://www.youtube.com/watch?v=6otW6OXjR8c. [Accessed: May 18, 2022]

[10] Ryan, "Answer to "how do i detect whether a variable is a function?". Stack overflow," Mar. 09, 2009. [Online]. Available: https://stackoverflow.com/a/624948/2859168. [Accessed: Jan. 10, 2023]

[11] N. Matloff, *The art of r programming: A tour of statistical software design*. No Starch Press, 2011 [Online]. Available: https://books.google.com/books?id=o2aLBAAAQBAJ

[12] M. Fripp, "Answer to "python pandas dataframe, is it pass-by-value or pass-by-reference". Stack overflow," Aug. 12, 2016. [Online]. Available: https://stackoverflow.com/a/38925257/2859168. [Accessed: Jan. 10, 2023]

[13] MathIsFun.com, "How to multiply matrices. Math is fun," 2021. [Online]. Available: https://www.mathsisfun.com/algebra/matrix-multiplying.html. [Accessed: Jan. 10, 2023]

[14] G. Grolemund and H. Wickham, *R for Data Science*, 1st ed. O'Reilly Media, 2017 [Online]. Available: https://r4ds.had.co.nz/. [Accessed: May 09, 2022]

[15] H. Wickham, *Advanced R*, 2nd ed. CRC Press, 2019 [Online]. Available: http://adv-r.had.co.nz/. [Accessed: May 09, 2022]

[16] Wikipedia Contributors, "Defensive programming," *Wikipedia*. Wikimedia Foundation, Apr. 2022 [Online]. Available: https://en.wikipedia.org/w/index.php?title=Defensive_programming&oldid=1084121123. [Accessed: May 31, 2022]

[17] A. Martelli and D. Ascher, *Python Cookbook*. O'Reilly Media, 2002 [Online]. Available: https://learning.oreilly.com/library/view/python-cookbook/0596001673/ch05s24.html. [Accessed: May 31, 2022]

[18] H. Wickham *et al.*, "Welcome to the tidyverse," *Journal of Open Source Software*, vol. 4, no. 43, p. 1686, 2019, doi: 10.21105/joss.01686.

[19] K. Ushey, *Renv: Project environments*. 2022 [Online]. Available: https://CRAN.R-project.org/package=renv

[20] H. Wickham and J. Bryan, *R Packages: Organize, Test, Document, and Share Your Code*, 1st ed. Sebastopol, CA: O'Reilly, 2015 [Online]. Available: https://r-pkgs.org/. [Accessed: Sep. 23, 2022]

[21] T. Beuzen and T. Timbers, *Python Packages*, 1st edition. Boca Raton: Chapman; Hall/CRC, 2022 [Online]. Available: https://py-pkgs.org/

[22] J. Evans, "A debugging manifesto https://t.co/3eSOFQj1e1," *Twitter*. Sep. 2022 [Online]. Available: https://twitter.com/b0rk/status/1570060516839641092. [Accessed: Sep. 21, 2022]

[23] Nasser_Junior, "User.fist_name https://t.co/lxrf3IFO4x," *Twitter*. Aug. 2020 [Online]. Available: https://twitter.com/Nasser_Junior/status/1295805928315531264. [Accessed: Sep. 21, 2022]

[24] J. Evans, "Debugging strategy: Reread the error message https://t.co/2BZHhPg04h," *Twitter*. Sep. 2022 [Online]. Available: https://twitter.com/b0rk/status/1570463473011920897. [Accessed: Sep. 21, 2022]

[25] J. Evans, "Debugging strategy: Shorten your feedback loop https://t.co/1cByDlafsK," *Twitter*. Jul. 2022 [Online]. Available: https://twitter.com/b0rk/status/1549164800978059264. [Accessed: Sep. 21, 2022]

[26] J. Evans, "Debugging strategy: Write a tiny program https://t.co/Kajr5ZyeIp," *Twitter*. Jul. 2022 [Online]. Available: https://twitter.com/b0rk/status/1547247776001654786. [Accessed: Sep. 21, 2022]

[27] J. Evans, "Debugging strategy: Change working code into broken code https://t.co/1T5uNDDFs0," *Twitter*. Jul. 2022 [Online]. Available: https://twitter.com/b0rk/status/1545099244238946304. [Accessed: Sep. 21, 2022]

[28] J. Evans, "Debugging strategy: Come up with one question https://t.co/2Lytzl4laQ," *Twitter*. Aug. 2022 [Online]. Available: https://twitter.com/b0rk/status/1554120424602193921. [Accessed: Sep. 21, 2022]

[29] J. Evans, "Debugging strategy: Write a unit test https://t.co/mC01DBNyM3," *Twitter*. Aug. 2022 [Online]. Available: https://twitter.com/b0rk/status/1561718747504803842. [Accessed: Sep. 21, 2022]

[30] T. Monteiro, "Improve how you code: Understanding rubber duck debugging. Duckly blog," Oct. 31, 2019. [Online]. Available: https://duckly.com/blog/improve-how-to-code-with-rubber-duck-debugging/. [Accessed: Jan. 11, 2023]

[31] S. Grimes, "This 500-Year-Old Piece of Advice Can Help You Solve Your Modern Problems," *Forge*. Dec. 2019 [Online]. Available: https://forge.medium.com/the-500-year-old-piece-of-advice-that-will-change-your-life-1e580f115731. [Accessed: Sep. 21, 2022]

[32] Quartz25, Jesdisciple, H. Röst, D. Ross, L. D'Oliveiro, and BLibrestez55, *Python Programming*. Wikibooks, 2016 [Online]. Available: https://en.wikibooks.org/wiki/Python_Programming. [Accessed: May 28, 2022]

[33] H. Wickham, "Tidy data," *The Journal of Statistical Software*, vol. 59, 2014 [Online]. Available: http://www.jstatsoft.org/v59/i10/

[34] DataMentor, "R recursion. DataMentor," Nov. 24, 2017. [Online]. Available: https://www.datamentor.io/r-programming/recursion/. [Accessed: Jan. 10, 2023]

[35] Parewa Labs Pvt. Ltd., "Python recursion. Learn python interactively," 2020. [Online]. Available: https://www.programiz.com/python-programming/recursion. [Accessed: Jan. 10, 2023]