

Introduction to Statistical Computing

Susan Vanderplas

Spring 2022

On this page

Preface	4
How to Use This Book	4
Special Sections	4
About This Book	6
1 Getting Started	7
Objectives	7
1.1 Computer Basics	7
1.1.1 Hardware	7
1.1.2 Operating Systems	8
1.1.3 File Systems	8
2 Setting up your computer	9
3 Scripts and Notebooks	11
3.1 Scripts	11
3.2 Notebooks	14
4 Finding your way in R and Python	17
4.1 Programming	17
4.2 Hello world	17
4.3 Talking to Python and R - Interactive mode	19
4.3.1 The R Console	21
4.3.2 The Python Console	23
4.4 Talking to Python and R - Script Mode	25
4.4.1 Comparing Python and R	41
4.5 Getting help	42
5 Basic Data Types	43
5.1 Values and Types	43
5.2 Variables	44
5.2.1 Valid Names	45
5.3 Type Conversions	46
5.4 Operators and Functions	49
5.4.1 Order of Operations	50
5.4.2 String Operations in Python	51

5.4.3	Functions	52
6	Data and Control Structures	53
6.1	Vectors	53
6.1.1	Indexing by Location	54
6.1.2	Indexing with Logical Vectors	57
6.1.3	Reviewing Types	59
6.2	Matrices	60
6.2.1	Indexing in Matrices	63
7	Data Structures	64
8	Reading in Data	65
	References	66

Preface

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.

How to Use This Book

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

Special Sections

Watch Out

Watch out sections contain things you may want to look out for - common errors, etc.

Examples

Example sections contain code and other information. Don't skip them!

My Opinion

These sections contain things you should definitely not consider as fact and should just take with a grain of salt.

Go Read

Sometimes, there are better resources out there than something I could write myself. When you see this section, go read the enclosed link as if it were part of the book.

Try It Out

Try it out sections contain activities you should do to reinforce the things you’ve just read.

Learn More

Learn More sections contain other references that may be useful on a specific topic. Suggestions are welcome (email me to suggest a new reference that I should add), as there’s no way for one person to catalog all of the helpful programming resources on the internet!

Note

Note sections contain clarification points (anywhere I would normally say “note that ...”)

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.

Many times, examples will be in expandable sections

This keeps the code and output from obscuring the actual information in the textbook that I want you to retain. You can always look up the syntax, but you do need to absorb the details I've written out.

About This Book

This is a Quarto book. To learn more about Quarto books visit <https://quarto.org/docs/books>.

I have written this entire book using reproducible techniques, with R and python code and results included within the book's text.

Stat 151 will be offered for the first time in Spring 2022, as I'm writing this in Fall 2021. Initially, my goal is to write the book in R and include python as an additional option/example. Eventually, I hope to teach Stat 151 in R and Python at the same time.

1 Getting Started

Objectives

1. Understand the basics of how computers work
2. Understand the file system mental model for computers
3. Set up RStudio, R, Quarto, and python
4. Be able to run demo code in R and python

1.1 Computer Basics

It is helpful when teaching a topic as technical as programming to ensure that everyone starts from the same basic foundational understanding and mental model of how things work. When teaching geology, for instance, the instructor should probably make sure that everyone understands that the earth is a round ball and not a flat plate – it will save everyone some time later.

We all use computers daily - we carry them around with us on our wrists, in our pockets, and in our backpacks. This is no guarantee, however, that we understand how they work or what makes them go.

1.1.1 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

1.1.2 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.

1.1.3 File Systems

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](#) 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 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.

2 Setting up your computer

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](#)

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.

In class activity

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 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.
- [Additional instructions for installing Python 3](#) from Python for Everybody

3 Scripts and Notebooks

In this class, we'll be using markdown notebooks to keep our code and notes in the same place. One of the advantages of both R and Python is that they are both scripting languages, but they can be used within notebooks as well. This means that you can have an R script file or a python script file, and you can run that file, but you can also create a document (like the one you're reading now) that has code AND text together in one place. This is called [literate programming](#) and it is a very useful workflow both when you are learning programming and when you are working as an analyst and presenting results.

3.1 Scripts

Before I show you how to use literate programming, let's look at what it replaces: scripts. **Scripts** are files of code that are meant to be run on their own. They may produce results, or format data and save it somewhere, or scrape data from the web – scripts can do just about anything.

Scripts can even have documentation within the file, using `#` characters (at least, in R and python) at the beginning of a line. `#` indicates a comment – that is, that the line does not contain code and should be ignored by the computer when the program is run. Comments are incredibly useful to help humans understand what the code does and why it does it.

3.1.0.1 Plotting a [logarithmic spiral](#) in R and python

This code will use concepts we have not yet introduced - feel free to tinker with it if you want, but know that you're not responsible for being able to **write** this code yet. You just need to read it and get a sense for what it does. I have heavily commented it to help with this process.

```
# Define the angle of the spiral (polar coords)
# go around two full times (2*pi = one revolution)
theta <- seq(0, 4*pi, .01)
# Define the distance from the origin of the spiral
# Needs to have the same length as theta
r <- seq(0, 5, length.out = length(theta))
```

```
# Now define x and y in cartesian coordinates
x <- r * cos(theta)
y <- r * sin(theta)

plot(x, y, type = "l")
```

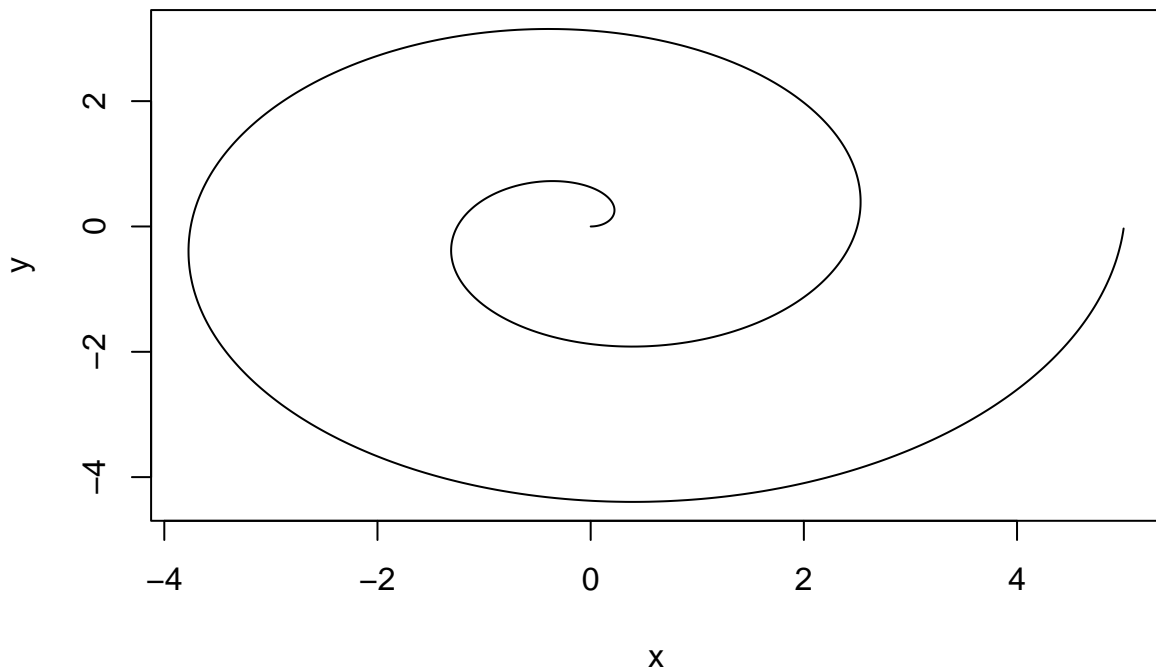


Figure 3.1: A Cartesian Spiral in R

I have saved this script [here](#). You can download it and open it up in RStudio (File -> Open -> Navigate to file location).

```
import numpy as np
import matplotlib.pyplot as plt

# Define the angle of the spiral (polar coords)
# go around two full times (2*pi = one revolution)
theta = np.arange(0, 4 * np.pi, 0.01)
# Define the distance from the origin of the spiral
# Needs to have the same length as theta
# (get length of theta with theta.size,
#  and then divide 5 by that to get the increment)
r = np.arange(0, 5, 5/theta.size)
```

```
# Now define x and y in cartesian coordinates
x = r * np.cos(theta)
y = r * np.sin(theta)

# Define the axes
fig, ax = plt.subplots()
# Plot the line
ax.plot(x, y)
plt.show()
```

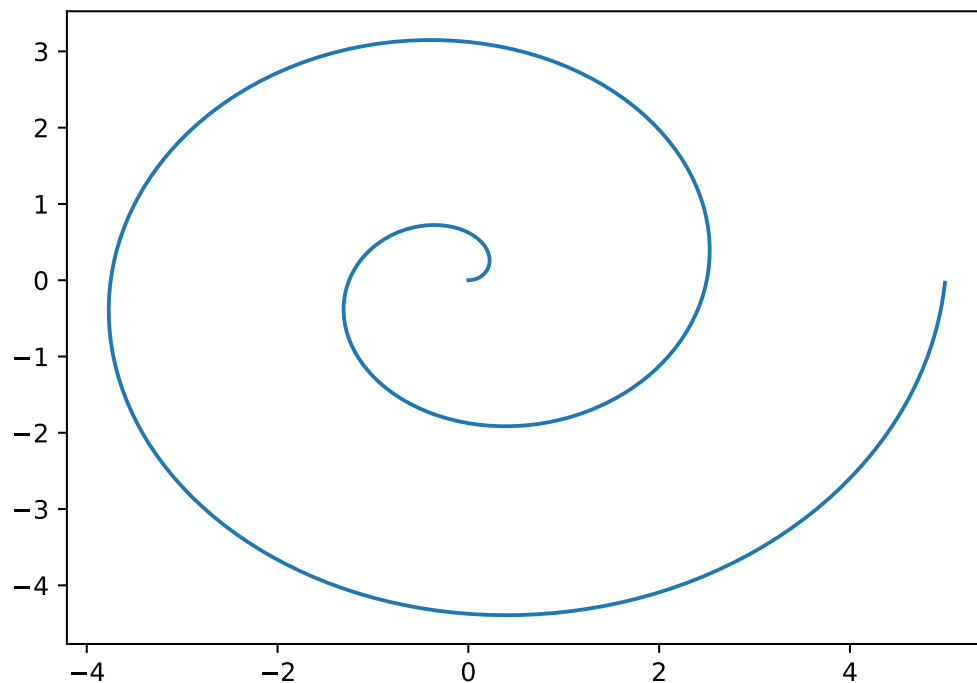


Figure 3.2: A Cartesian Spiral in python

I have saved this script [here](#). You can download it and open it up in RStudio (File -> Open -> Navigate to file location).

Scripts can be run in Rstudio by clicking the Run button  at the top of the editor window when the script is open.

3.1.0.2 Try it out!

- Download the R and python scripts in the above example, open them in RStudio, and run each script using the Run button. What do you see?
- (Advanced) Open a terminal in RStudio (Tools -> Terminal -> New Terminal) and see if you can run the R script from the terminal using `R CMD BATCH path/to/file/markdown-spiral-script.R` (You will have to modify this command to point to the file on your machine)
Notice that two new files appear in your working directory: `Rplots.pdf` and `markdown-spiral-script.Rout`
- (Advanced) Open a terminal in RStudio (Tools -> Terminal -> New Terminal) and see if you can run the R script from the terminal using `python3 path/to/file/markdown-spiral-script.py` (You will have to modify this command to point to the file on your machine)
This will require you to have python3 accessible to you on the command line, which may be a challenge if it is not set up in the way that I'm assuming it is. Feel free to make an appointment to see if we can figure it out, if this does not work the first time.

Most of the time, you will run scripts interactively - that is, you'll be sitting there watching the script run and seeing what the results are as you are modifying the script. However, one advantage to scripts over notebooks is that it is easy to write a script and schedule it to run without supervision to complete tasks which may be repetitive. I have a script that runs daily at midnight, 6am, noon, and 6pm to pull information off of the internet for a dataset I'm maintaining. I've set it up so that this all happens automatically and I only have to check the results when I am interested in working with that data.

3.2 Notebooks

Notebooks are an implementation of literate programming. Both R and python have native notebooks that are cross-platform and allow you to code in R or python. This book is written using Quarto markdown, which is an extension of Rmarkdown, but it is also possible to use jupyter notebooks to write R code.

In this class, we're going to use Quarto/R markdown, because it is a much better tool for creating polished reports than Jupyter (in my opinion). This matters because the goal is that you learn something useful for your own coding and then you can easily apply it when you go to work as an analyst somewhere to produce impressive documents. Jupyter notebooks are great for interactive coding, but aren't so wonderful for producing polished results. They also don't allow you to switch between languages mid-notebook, and since I'm trying to teach this class in both R and python, I want you to have both languages available.

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`.

3.2.0.1 Try it out - R markdown

Take a look at the [R markdown sample file](#) I've created to go with the R script above. You can see the HTML file it generates [here](#).

- Download the Rmd file and open it with RStudio.
- Change the output to `output: word_document` and hit the Render button



. Can you find the markdown-demo.docx file that was generated? What does it look like?

- Change the output to `output: pdf_document` and hit the Render button



Can you find the markdown-demo.pdf file that was generated? What does it look like?

Rmarkdown tries very hard to preserve your formatted text appropriately regardless of the output document type. While things may not look exactly the same, the goal is to allow you to focus on the content and the formatting will “just work”.

3.2.0.2 Try it out - Jupyter

Take a look at the [jupyter notebook sample file](#) I've created to go with the R script above. You can see the HTML file it generates [here](#).

- Download the ipynb file and open it with jupyter.
- Export the notebook as a pdf file (File -> Save as -> PDF via HTML). Can you find the jupyter-demo.pdf file that was generated? What does it look like?
- Export the notebook as an html file (File -> Save as -> HTML). Can you find the jupyter-demo.html file that was generated? What does it look like?

3.2.0.3 Try it out - Quarto markdown

The nice thing about quarto is that it will work with python and R seamlessly, and you can compile the document using python or R. R markdown will also allow you to use python chunks, but you must compile the document in R.

Take a look at the [Qmd notebook sample file](#) I've created to go with the scripts above. You'll notice that it is basically the script portion of this textbook – that's because I'm writing the textbook in Quarto.

- Download the qmd file and open it with RStudio

- Try to compile the file by hitting the Render button



- (Advanced) In the terminal, type in `quarto render path/to/file/quarto-demo.qmd`. Does that render the HTML file?

One advantage of this is that using quarto to render the file doesn't require R at the command line. As the document contains R chunks, R is still required to compile the document, but the biggest difference between qmd and rmd is that qmd files are workflow agnostic - you can generate them in e.g. MS Visual Studio Code, compile them in that workflow, and never have to use RStudio.

4 Finding your way in R and Python

4.1 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 small problems
- communicate the solution to a computer using a programming language

In this class, 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 of the semester, 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.

4.2 Hello world

I particularly like the way that Python for Everybody (Severance 2016) 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, 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.

4.3 Talking to Python and R - Interactive mode

R and python both have an “interactive mode” that you will use most often. In the previous chapter, we talked about scripts and markdown documents, both of which are non-interactive methods for writing R and python code. But for the moment, let’s work with the interactive console in both languages in order to get familiar with how we talk to R and python.

Let’s start by creating a Qmd file (File -> New File -> Quarto Document) - this will let us work with R and python at the same time.

Add an R chunk to your file by typing ```{r}` into the first line of the file, and then hit return. RStudio should add a blank line followed by ````.

Add a python chunk to your file by typing ```{python}` on a blank line below the R chunk you just created, and then hit return. RStudio should add a blank line followed by ````.

Your file should look like this:

If instead your file looks like this:

you have visual markdown mode on. To turn it off, click on the A icon at the top right of your editor window:

If we are working in interactive mode, why did I have you start out by creating a markdown document? Good Question! RStudio allows you to switch back and forth between R and python seamlessly, which is good and bad - it’s hard to get a python terminal without telling R which language you’re working in! You can create a python script if you’d prefer to work in a script instead of a markdown document, but that would involve working in 2 separate files, which I personally find rather tedious.

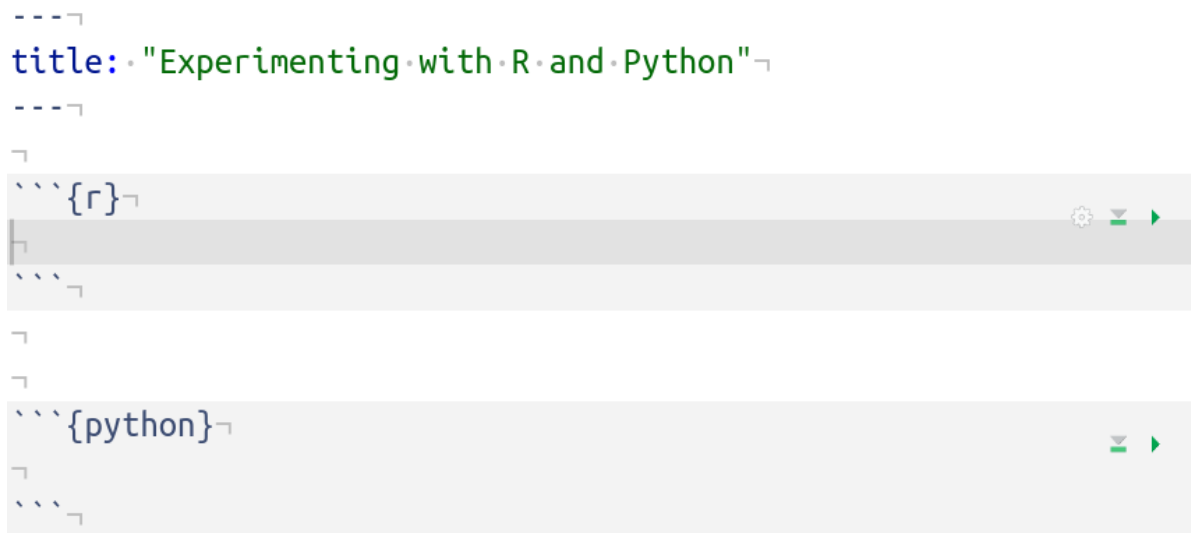


Figure 4.1: Screenshot of qmd file after adding an empty r and python chunk

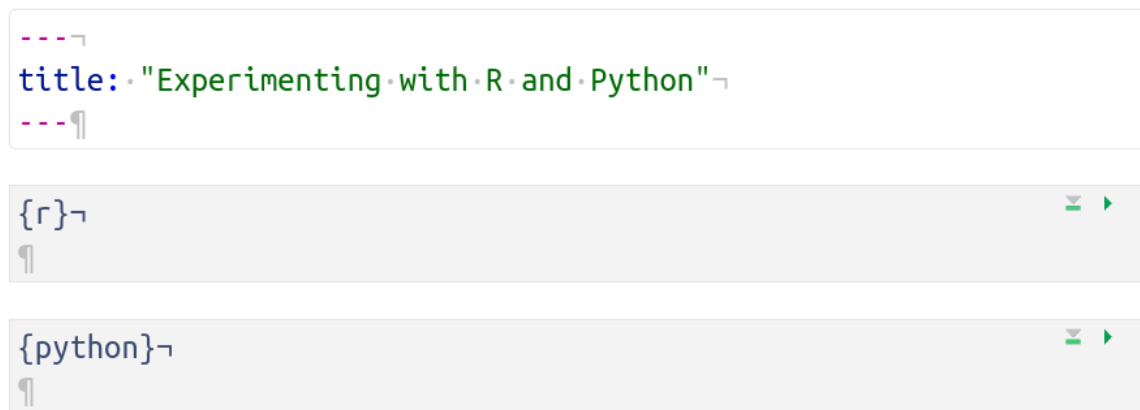


Figure 4.2: Screenshot of qmd file with visual markdown editing on



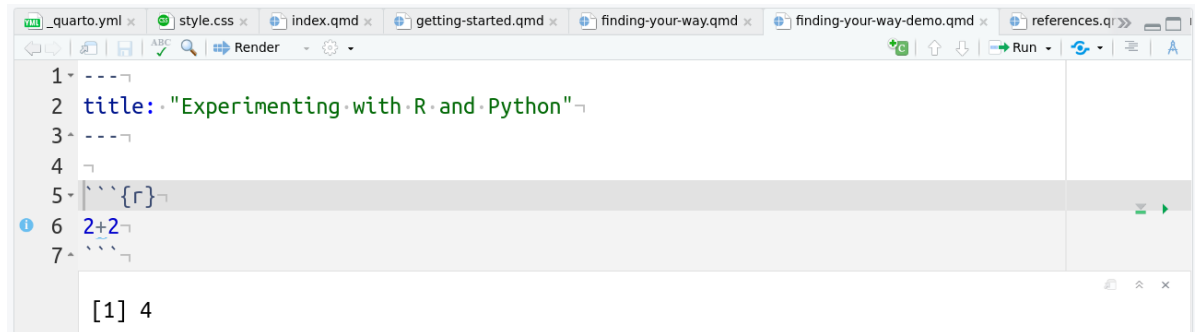
Figure 4.3: Screenshot of editor window toolbar, with A icon highlighted in green

4.3.1 The R Console


In your R chunk or script, type in `2+2` and hit **Ctrl+Enter** (or **Cmd+Enter** on a mac). Look down to the Console (which is usually below the editor window) and see if 4 appears. If you're like me, output shows up in two places at once:

Location Picture

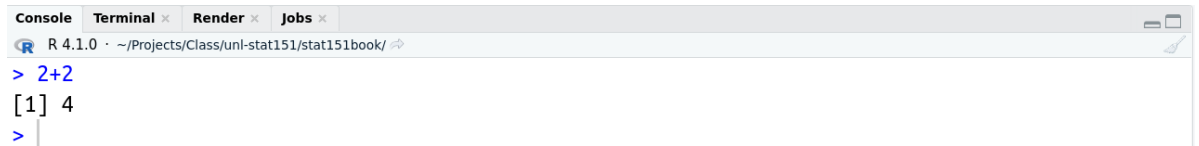
Chunk



Script



Console

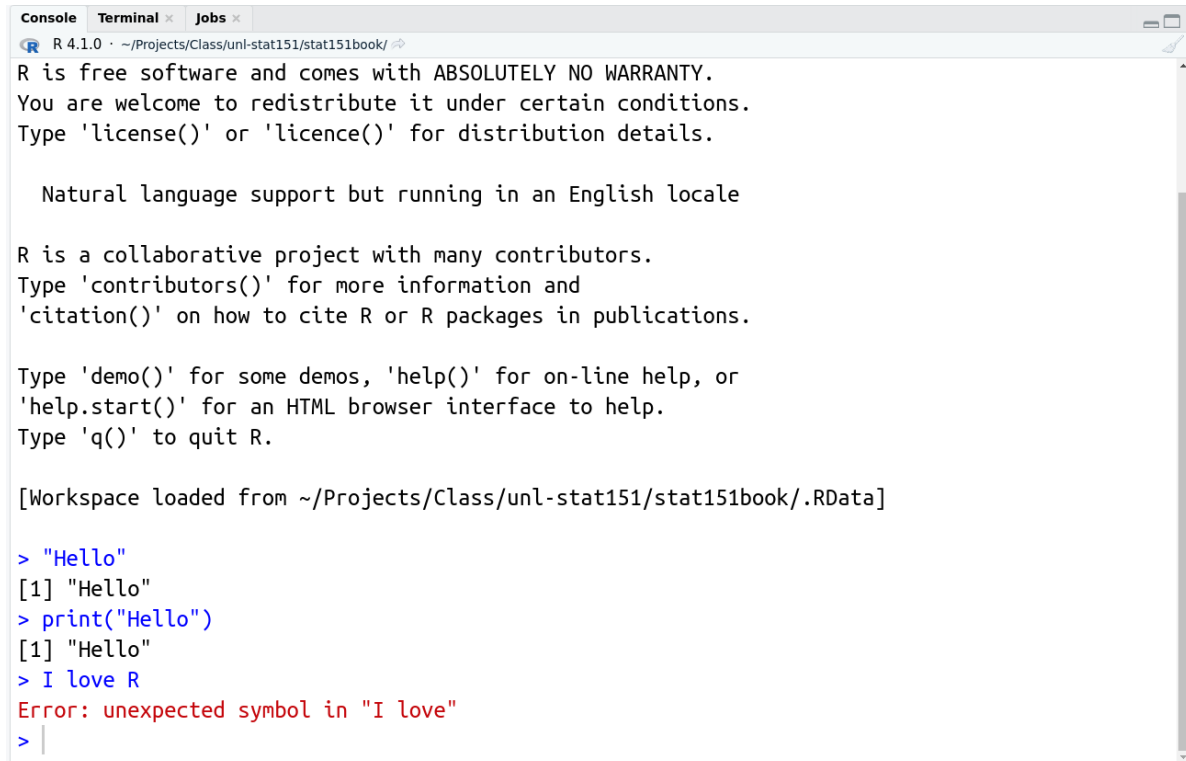


R will indicate that it is waiting for your command with a `>` character in the console. If you don't see that `>` character, chances are you've forgotten to finish a statement - check for parentheses and brackets.

When you are working in an R script, any output is shown only in the console. When you are working in an R code chunk, output is shown both below the chunk and in the console.

If you want, you can also just work within the R console. This can be useful for quick, interactive work, or if, like me, you're too lazy to pull up a calculator on your machine and you just want to use R to calculate something quickly. You just type your R command into the console:

The first two statements in the above example work - "Hello" is a string, and is thus a valid statement equivalent to typing "2" into the console and getting "2" back out. The second command, `print("Hello")`, does the same thing - "Hello" is returned as the result. The third command, `I love R`, however, results in an error - there is an unexpected symbol (the space)



```
R 4.1.0 · ~/Projects/Class/unl-stat151/stat151book/
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

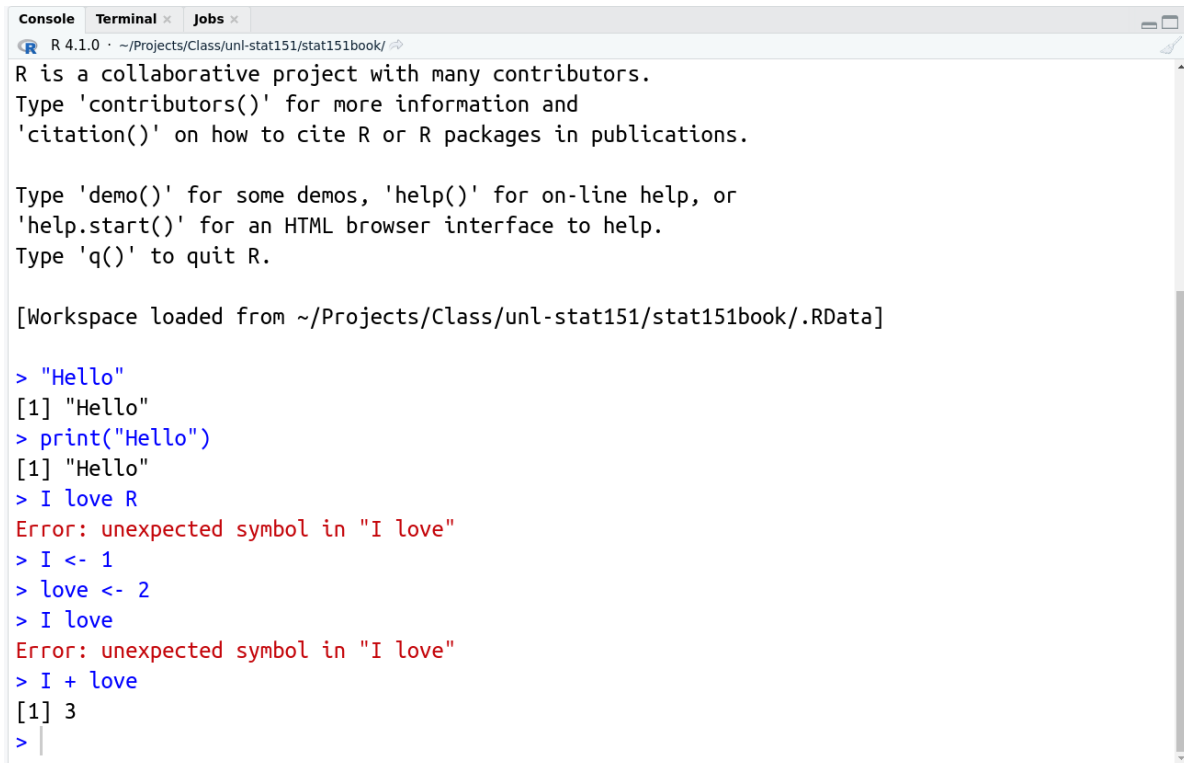
[Workspace loaded from ~/Projects/Class/unl-stat151/stat151book/.RData]

> "Hello"
[1] "Hello"
> print("Hello")
[1] "Hello"
> I love R
Error: unexpected symbol in "I love"
> |
```

Figure 4.4: R console with commands “Hello”, `print(“Hello”)`, and (unquoted) “I love R”, which causes an error

in the statement. R thinks we are telling it to do something with variables `I` and `love` (which are not defined), and it doesn't know what we want it to do to the two objects.

Suppose we define `I` and `love` as variables by putting a value into each object using `<-`, which is the assignment operator. Then, typing “I love” into the console generates the same error, and R tells us “hey, there's an unexpected symbol here” - in this case, maybe we meant to add the two variables together.



```
R 4.1.0 · ~/Projects/Class/unl-stat151/stat151book/
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

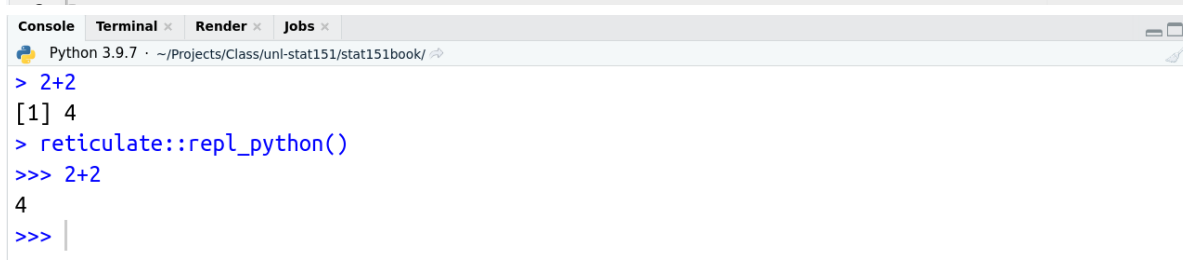
[Workspace loaded from ~/Projects/Class/unl-stat151/stat151book/.RData]

> "Hello"
[1] "Hello"
> print("Hello")
[1] "Hello"
> I love R
Error: unexpected symbol in "I love"
> I <- 1
> love <- 2
> I love
Error: unexpected symbol in "I love"
> I + love
[1] 3
>
```

Figure 4.5: R console with commands “Hello”, `print(“Hello”)`, and (unquoted) “I love R”, which causes an error. Defining variables `I` and `love` provides us a context in which R’s error message about unexpected symbols makes sense - R is reminding us that we need a numerical operator in between the two variable names.


4.3.2 The Python Console

In your python chunk or script, type in `2+2` and hit **Ctrl+Enter** (or **Cmd+Enter** on a mac). Look down to the Console (which is usually below the editor window) and see if 4 appears. If you’re like me, output shows up in two places at once:

<p>Chunk</p>	
<p>Script</p>	
<p>Console</p>	

Notice that in the console, you get a bit of insight into how RStudio is running the python code: we see `reticulate::repl_python()`, which is R code telling R to run the line in Python. The python console has `>>>` instead of `>` to indicate that python is waiting for instructions.

Notice also that the only difference between the R and python script file screenshots is that

there is a different logo on the documents: . Personally, I think it's easier to work in a markdown document and keep my notes with specific chunks labeled by language when I'm learning the two languages together, but when you are writing code for a specific project in a single language, it is probably better to use a script file specific to that language.

If you want to start the python console by itself (without a script or working in a markdown document), you can simply type `reticulate::repl_python()` into the R console.


```
Python 3.9.7 · ~/Projects/Class/unl-stat151/stat151book/
> reticulate::repl_python()
Python 3.9.7 (/home/susan/.local/share/r-miniconda/envs/r-reticulate/bin/python)
Reticulate 1.22 REPL -- A Python interpreter in R.
Enter 'exit' or 'quit' to exit the REPL and return to R.
>>> |
```

R is nice enough to remind you that to end the conversation with python, you just need to type “exit” or “quit”.

If you want to start a python console outside of RStudio, bring up your command prompt (Darwin on mac, Konsole on Linux, CMD on Windows) and type `python3` into that window and you should see the familiar `>>>` waiting for a command.

4.4 Talking to Python and R - Script Mode

In the last chapter, we played around with scripts and markdown documents for python and R. In the last section, we played with interactive mode by typing R and python commands into a console or running code chunks interactively using the Run button or `Ctrl/Cmd + Enter` (which is the keyboard shortcut).

You may be learning to program in R and python because it's a required part of the curriculum, but hopefully, you also have some broader ideas of what you might do with either language - process data, make pretty pictures, write a program to trigger the computer uprising...

Scripts are best used when you have a thing you want to do, and you will need to do that thing many times, perhaps with different input data. Suppose that I have a text file and I want to pull out the most common word in that file. In the next few examples, I will show you how to do this in R and python, and at the same time, demonstrate the difference between interactive mode and script mode in both languages. In each example, try to compare to the previous example to identify whether something is running as a full script or in interactive mode, and how it is launched (in R? at the command line?).

Severance (2016) provides a handy python program to count words. This program is meant to be run on the command line, and it will run for any specified text file.

4.4.0.1 Example: Counting Words in Python on the Command Line

Download [words.py](#) to your computer and open up a command line terminal in the location where you saved the file.

Before you run the script, save [Oliver Twist](#) to the same folder as `dickens-oliver-627.txt` (you can use another file name, but you will have to adjust your response to the program)

```

name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py

```

In your terminal, type in `python words.py`. If all goes well, you should get a prompt that tells you to enter a file name. Type in `dickens-oliver-627.txt`, and the program will read in the file and execute the program according to the instructions shown above. You don't need to understand what is happening in this program (just like you don't need to understand what is happening in the R code above either) – you get the answer anyways: the most common word, according to the output from the program, is

`the 8854`

That is, the word `the` occurs 8854 times in the text.

4.4.0.2 Example: Counting Words in Python within RStudio

We can run this script in interactive mode in RStudio if we want to: Open the `words.py` file you downloaded in RStudio.

Click the “Source Script” button highlighted in green above, and then look at the console below the script window:

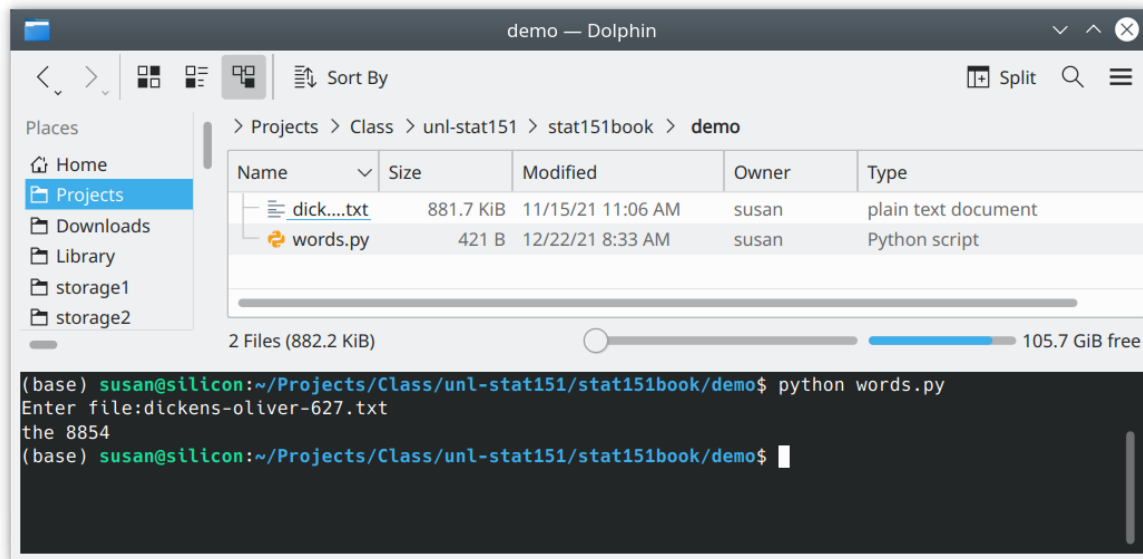
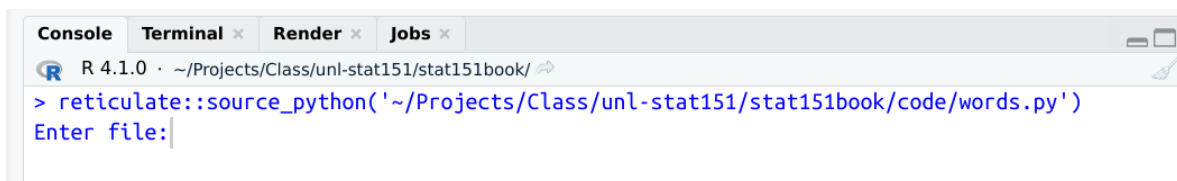


Figure 4.6: Screenshot of folder and python script evaluation, showing how to run the python script in the terminal and get the count of the most common word, ‘the’, in the file dickens-oliver-627.txt



Once you enter the path to the text file – this time, from the **project** working directory – you get the same answer. It can be a bit tricky to figure out what your current working directory is in RStudio, but in the R console you can get that information with the `getwd()` command.

Since I know that I have stored the text file in the **data** subdirectory of the **stat151book** folder, I can type in `data/dickens-oliver-627.txt` and the python program can find my file.

In the above example, RStudio is functioning essentially like a terminal window - it runs the script as a single file, and once it has your input, all commands are executed one after the other automatically. This is convenient if you want to test the whole block of code at once, but it can be more useful to test each line individually and “play” with the output a bit (or modify code line-by-line).

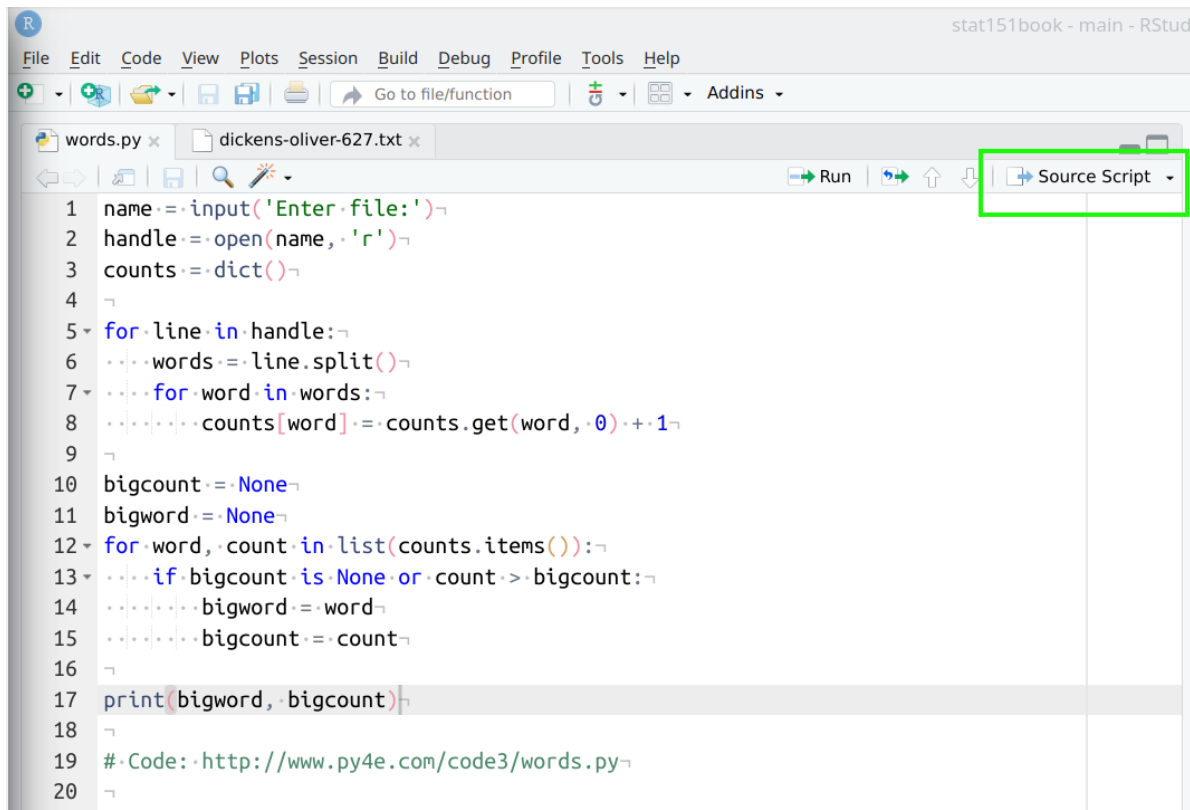


Figure 4.7: Rstudio screenshot showing the words.py file opened, with a green highlighted rectangle around the button “Source Script” which allows you to run the file in RStudio.

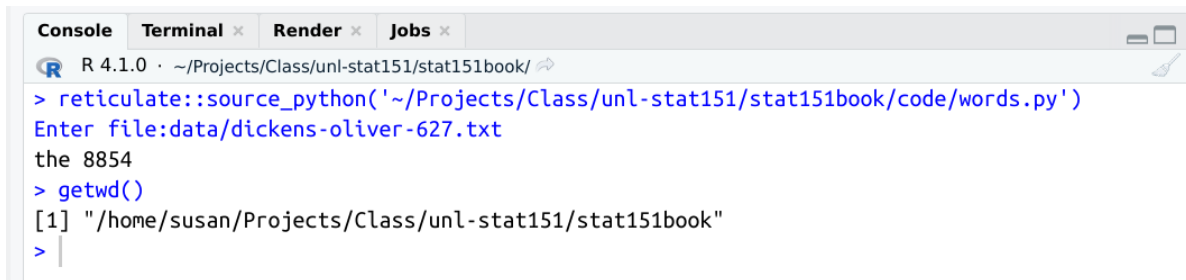


Figure 4.8: RStudio screenshot of console window with `getwd()` command and result

4.4.0.3 Example: Counting Words in Python in Interactive Mode (RStudio)

Suppose we want to modify this python script to be more like the R script, where we tell python what the file name is in the file itself, instead of waiting for user input at the terminal.

Instead of using the `input` command, I just provide python with a string that contains the path to the file. If you have downloaded the text file to a different folder and RStudio's working directory is set to that folder, you would change the first line to `name = "dickens-oliver-627.txt"` - I have set things up to live in a data folder because if I had all of the files in the same directory where this book lives, I would never be able to find anything.

Create a new python script file in RStudio (File -> New File -> Python Script) and paste in the following lines of code, adjusting the path to the text file appropriately.

```
name = "data/dickens-oliver-627.txt"
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

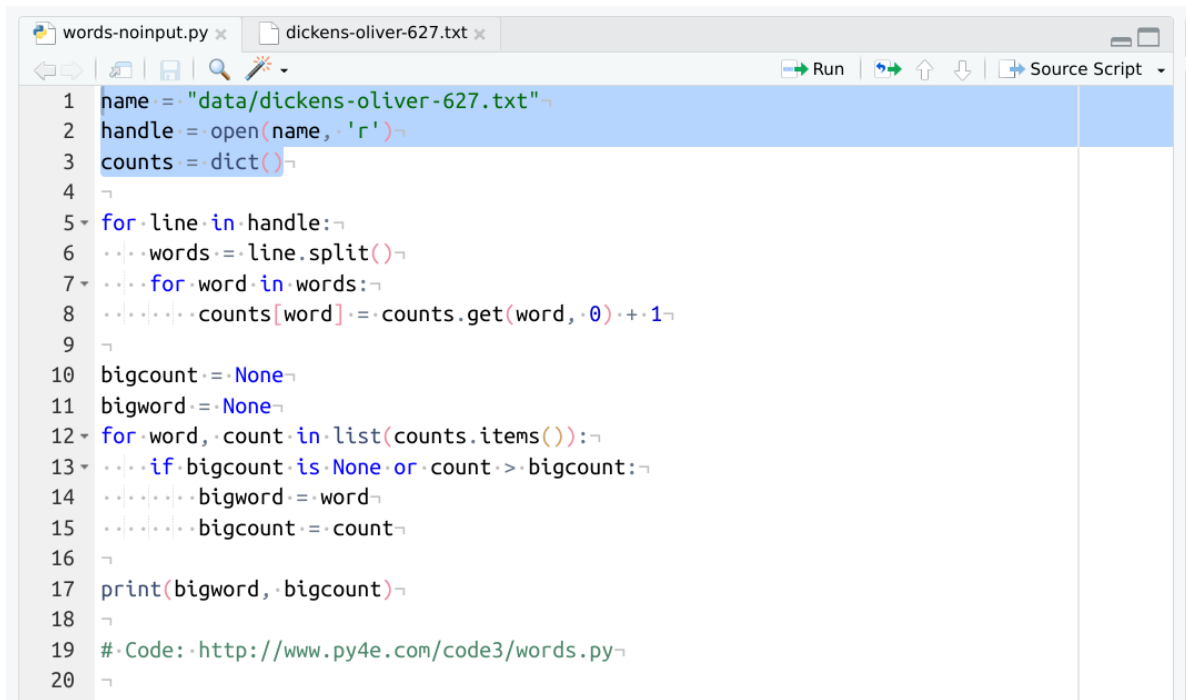
bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py
```

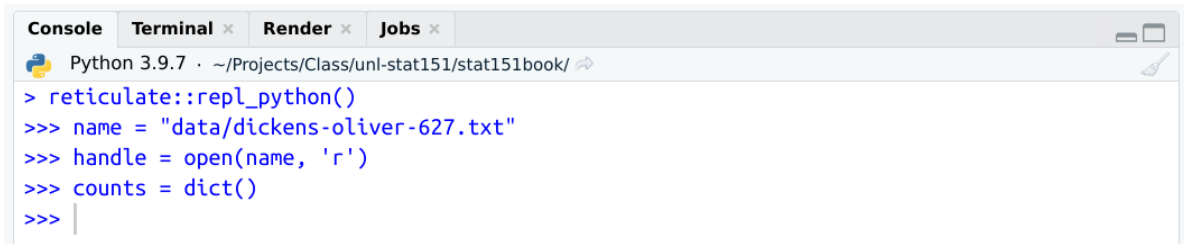
the 8854

With the first 3 lines highlighted, click the Run button.



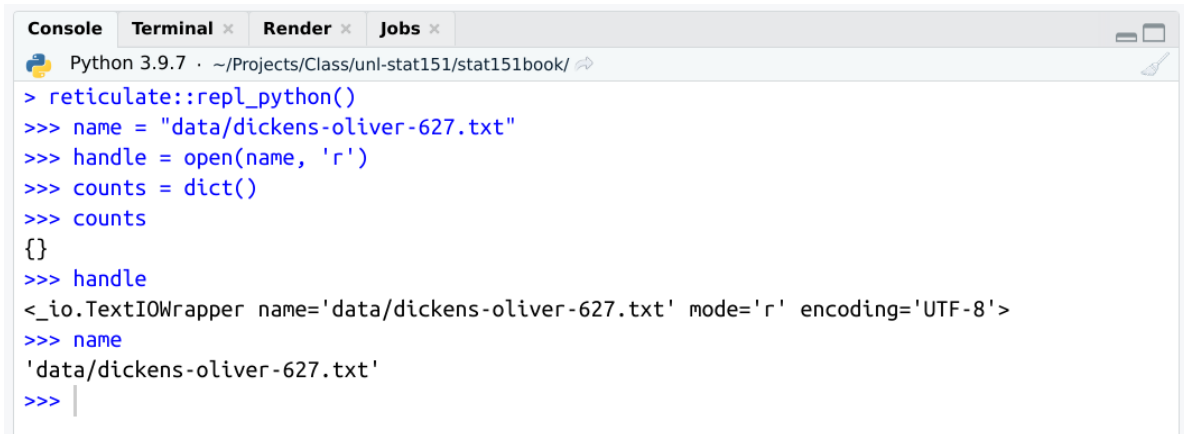
```
1 name = "data/dickens-oliver-627.txt"
2 handle = open(name, 'r')
3 counts = dict()
4
5 for line in handle:
6     words = line.split()
7     for word in words:
8         counts[word] = counts.get(word, 0) + 1
9
10 bigcount = None
11 bigword = None
12 for word, count in list(counts.items()):
13     if bigcount is None or count > bigcount:
14         bigword = word
15         bigcount = count
16
17 print(bigword, bigcount)
18
19 # Code: http://www.py4e.com/code3/words.py
20
```

Figure 4.9: Above script in the RStudio text editor window, with the first 3 lines of code highlighted



```
Python 3.9.7 · ~/Projects/Class/unl-stat151/stat151book/
> reticulate::repl_python()
>>> name = "data/dickens-oliver-627.txt"
>>> handle = open(name, 'r')
>>> counts = dict()
>>> |
```

We can examine the objects that we have defined this far in the program by typing their names into the console directly.



```
Python 3.9.7 · ~/Projects/Class/unl-stat151/stat151book/
> reticulate::repl_python()
>>> name = "data/dickens-oliver-627.txt"
>>> handle = open(name, 'r')
>>> counts = dict()
>>> counts
{}
>>> handle
<_io.TextIOWrapper name='data/dickens-oliver-627.txt' mode='r' encoding='UTF-8'>
>>> name
'data/dickens-oliver-627.txt'
>>> |
```

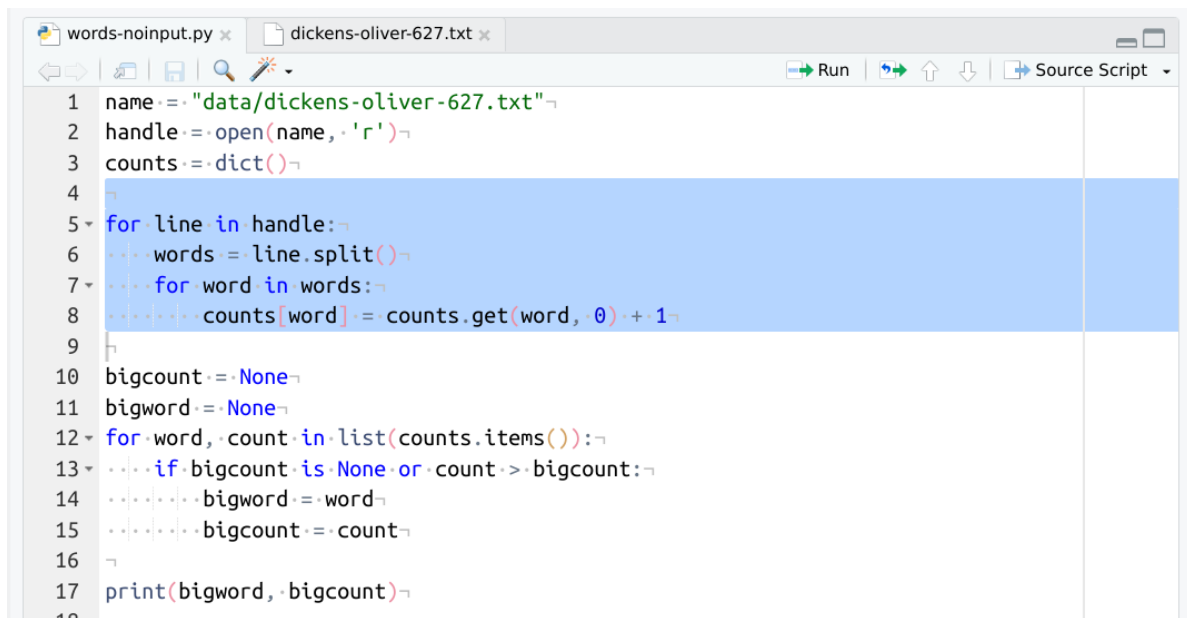
Figure 4.10: RStudio python console allows us to examine the objects we have defined after the first 3 lines of code have been run. We can see that `counts` is an empty object, `handle` is a pointer to a text file, and `name` is a string with the path to the text file – so far, so good.

If we want to continue walking through the program chunk by chunk, we can run the next four lines of code. Lines 5-8 are a for loop, so we should run them all at once unless we want to fiddle with how the for loop works.

Select lines 5-8 as shown above, and click the Run button. Your console window should update with additional lines of code. You can type in `counts` after that has been evaluated to see what the `counts` object looks like now.

The next few lines of code determine which word has the highest count. We won't get into the details here, but to finish out the running of the program, select lines 10-17 and run them in RStudio.

Running scripts in interactive mode or within RStudio is much more convenient if you are still working on the script - it allows you to debug the script line-by-line if necessary. Running a script at the terminal (like we did above) is sometimes more convenient if you have a pre-written script that you know already works. Both modes are useful, but for the time being you

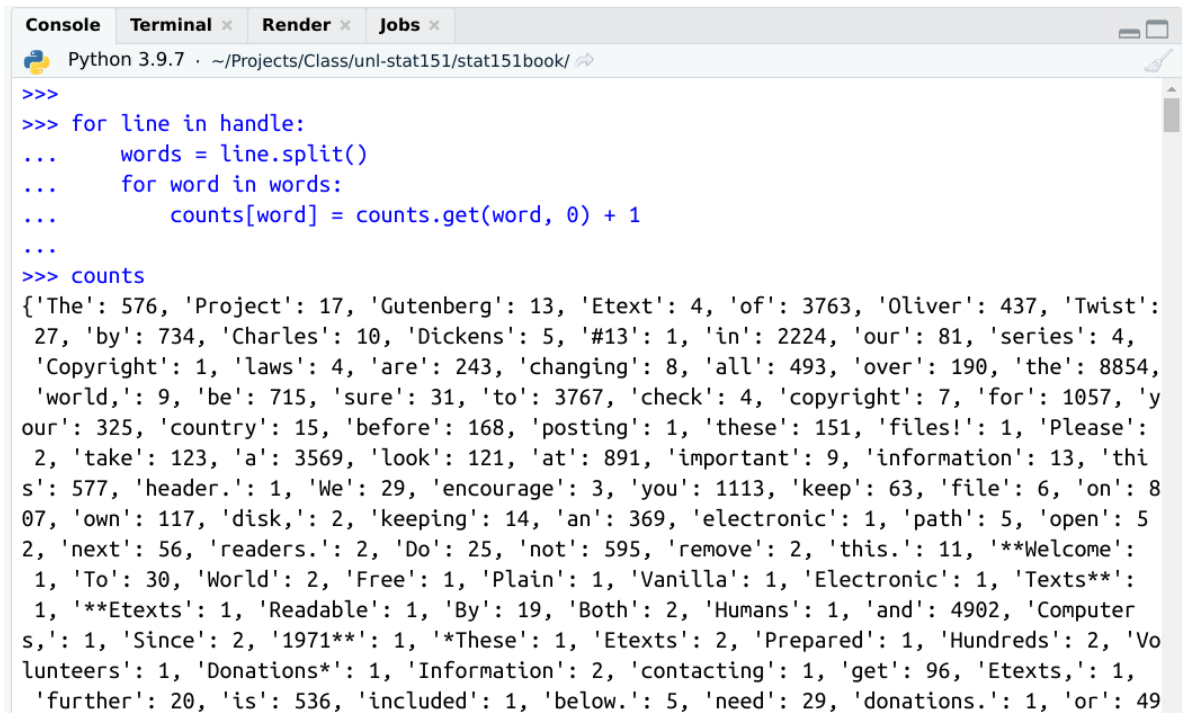
The image shows an RStudio editor window with two tabs: 'words-noinput.py' and 'dickens-oliver-627.txt'. The Python script in the 'words-noinput.py' tab contains 18 lines of code. Lines 5 through 8 are highlighted in blue. The code defines a file path, opens the file, creates a dictionary to count words, and iterates through the file to update the counts. The highlighted lines are:

```
5 for line in handle:
6     words = line.split()
7     for word in words:
8         counts[word] = counts.get(word, 0) + 1
```

The rest of the code in the file includes:

```
9
10 bigcount = None
11 bigword = None
12 for word, count in list(counts.items()):
13     if bigcount is None or count > bigcount:
14         bigword = word
15         bigcount = count
16
17 print(bigword, bigcount)
18 _
```

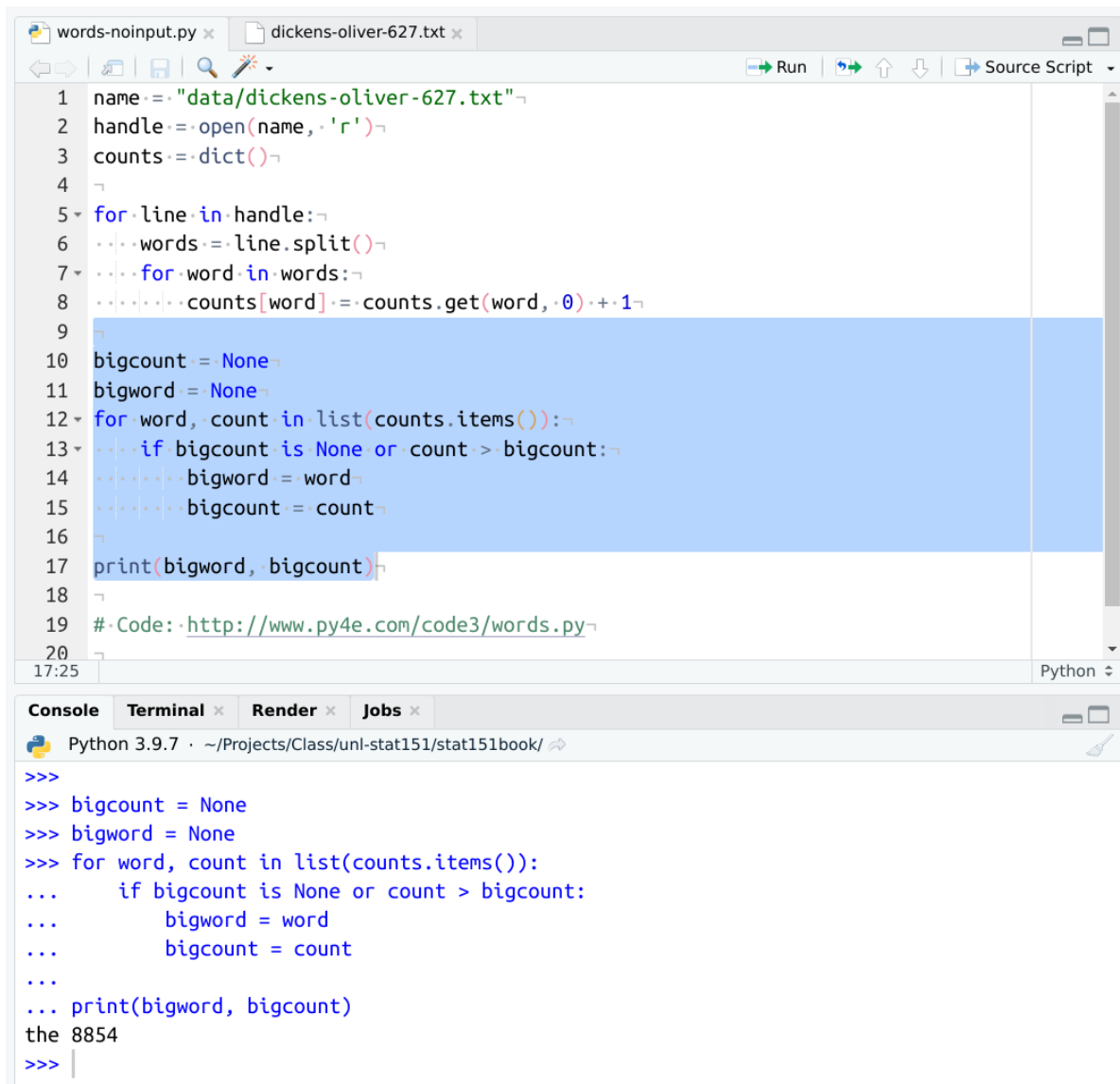
Figure 4.11: RStudio editor window with the next four lines of the code chunk highlighted. If we click the Run button, we can tell python to evaluate these few lines of code, and then we can see what the objects we’ve defined look like once that has been done



```
Console Terminal x Render x Jobs x
Python 3.9.7 · ~/Projects/Class/unl-stat151/stat151book/

>>>
>>> for line in handle:
...     words = line.split()
...     for word in words:
...         counts[word] = counts.get(word, 0) + 1
...
>>> counts
{'The': 576, 'Project': 17, 'Gutenberg': 13, 'Etext': 4, 'of': 3763, 'Oliver': 437, 'Twist': 27, 'by': 734, 'Charles': 10, 'Dickens': 5, '#13': 1, 'in': 2224, 'our': 81, 'series': 4, 'Copyright': 1, 'laws': 4, 'are': 243, 'changing': 8, 'all': 493, 'over': 190, 'the': 8854, 'world': 9, 'be': 715, 'sure': 31, 'to': 3767, 'check': 4, 'copyright': 7, 'for': 1057, 'y our': 325, 'country': 15, 'before': 168, 'posting': 1, 'these': 151, 'files!': 1, 'Please': 2, 'take': 123, 'a': 3569, 'look': 121, 'at': 891, 'important': 9, 'information': 13, 'this': 577, 'header.': 1, 'We': 29, 'encourage': 3, 'you': 1113, 'keep': 63, 'file': 6, 'on': 807, 'own': 117, 'disk.': 2, 'keeping': 14, 'an': 369, 'electronic': 1, 'path': 5, 'open': 52, 'next': 56, 'readers.': 2, 'Do': 25, 'not': 595, 'remove': 2, 'this.': 11, '**Welcome': 1, 'To': 30, 'World': 2, 'Free': 1, 'Plain': 1, 'Vanilla': 1, 'Electronic': 1, 'Texts**': 1, '**Etexts': 1, 'Readable': 1, 'By': 19, 'Both': 2, 'Humans': 1, 'and': 4902, 'Computers.': 1, 'Since': 2, '1971**': 1, '*These': 1, 'Etexts': 2, 'Prepared': 1, 'Hundreds': 2, 'Volunteers': 1, 'Donations*': 1, 'Information': 2, 'contacting': 1, 'get': 96, 'Etexts.': 1, 'further': 20, 'is': 536, 'included': 1, 'below.': 5, 'need': 29, 'donations.': 1, 'or': 49}
```

Figure 4.12: RStudio python console with lines 5-8 run and the counts object displayed. Counts is now filled with words and corresponding integer counts of the frequency of that word's appearance in the text



The image shows the RStudio interface with a Python script editor and a console window. The script, named 'words-noinput.py', is open and shows lines 1 through 20. Lines 10 through 17 are highlighted in blue, indicating they are the current focus. The console window at the bottom shows the output of the script, which is 'the 8854'.

```
1 name := "data/dickens-oliver-627.txt"
2 handle := open(name, 'r')
3 counts := dict()
4
5 for line in handle:
6     words := line.split()
7     for word in words:
8         counts[word] := counts.get(word, 0) + 1
9
10 bigcount := None
11 bigword := None
12 for word, count in list(counts.items()):
13     if bigcount is None or count > bigcount:
14         bigword = word
15         bigcount = count
16
17 print(bigword, bigcount)
18
19 # Code: http://www.py4e.com/code3/words.py
20
```

Console output:

```
>>>
>>> bigcount = None
>>> bigword = None
>>> for word, count in list(counts.items()):
...     if bigcount is None or count > bigcount:
...         bigword = word
...         bigcount = count
...
... print(bigword, bigcount)
the 8854
>>>
```

Figure 4.13: RStudio editor window and console showing the results when lines 10-17 are evaluated. It is clear that line 17 results in the console output of `the 8854`

will probably be running scripts within your development environment (RStudio or VSCode or any other IDE you prefer) more often than at the command line.

4.4.0.4 Example: Counting Words in R within RStudio

Just for fun, let's work with [Oliver Twist](#), by Charles Dickens, which I have saved [here](#).

```
# Read in the file
text <- readLines("dickens-oliver-627.txt")

# Split the lines of text into separate words
text <- strsplit(text, " ")

# Simplify the list
text <- unlist(text)

# Count up the number of occurrences of each word
word_freq <- table(text)

# Sort the table by decreasing frequency
word_freq <- sort(word_freq, decreasing = T)

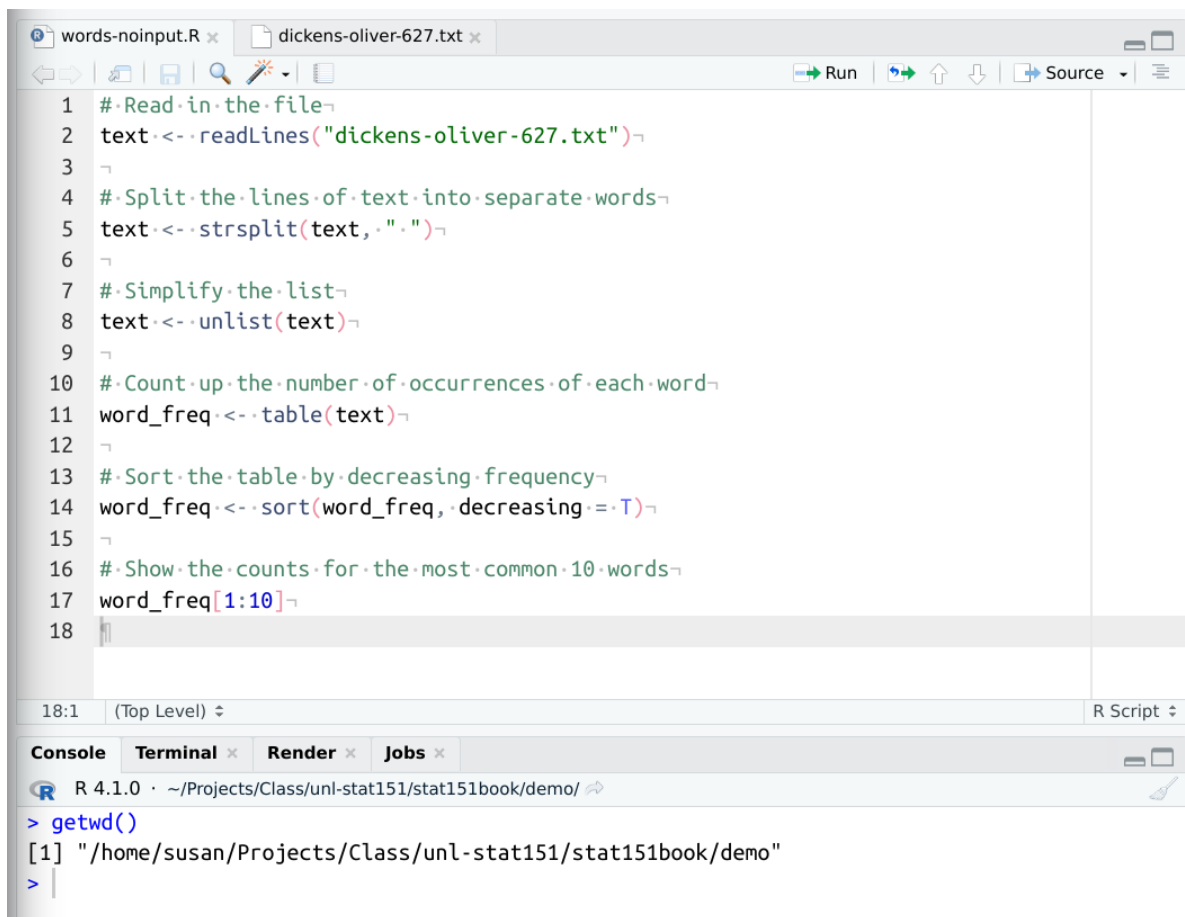
# Show the counts for the most common 10 words
word_freq[1:10]
```

Make a new R script (File -> New File -> R script) and copy the above code into R, or [download the file to your computer directly](#) and open the downloaded file in RStudio.

In the R console, run the command `getwd()` to see where R is running from. This is your “working directory”.

Save the copy of Oliver Twist to the file `dickens-oliver-627.txt` in the folder that `getwd()` spit out. You can test that you have done this correctly by typing `list.files()` into the R console window and hitting enter. It is very important that you know where on your computer R is looking for files - otherwise, you will constantly get “file not found” errors, and that will be very annoying.

Use the “Run” button to run the script and see what the output is. How many times does ‘the’ appear in the file?



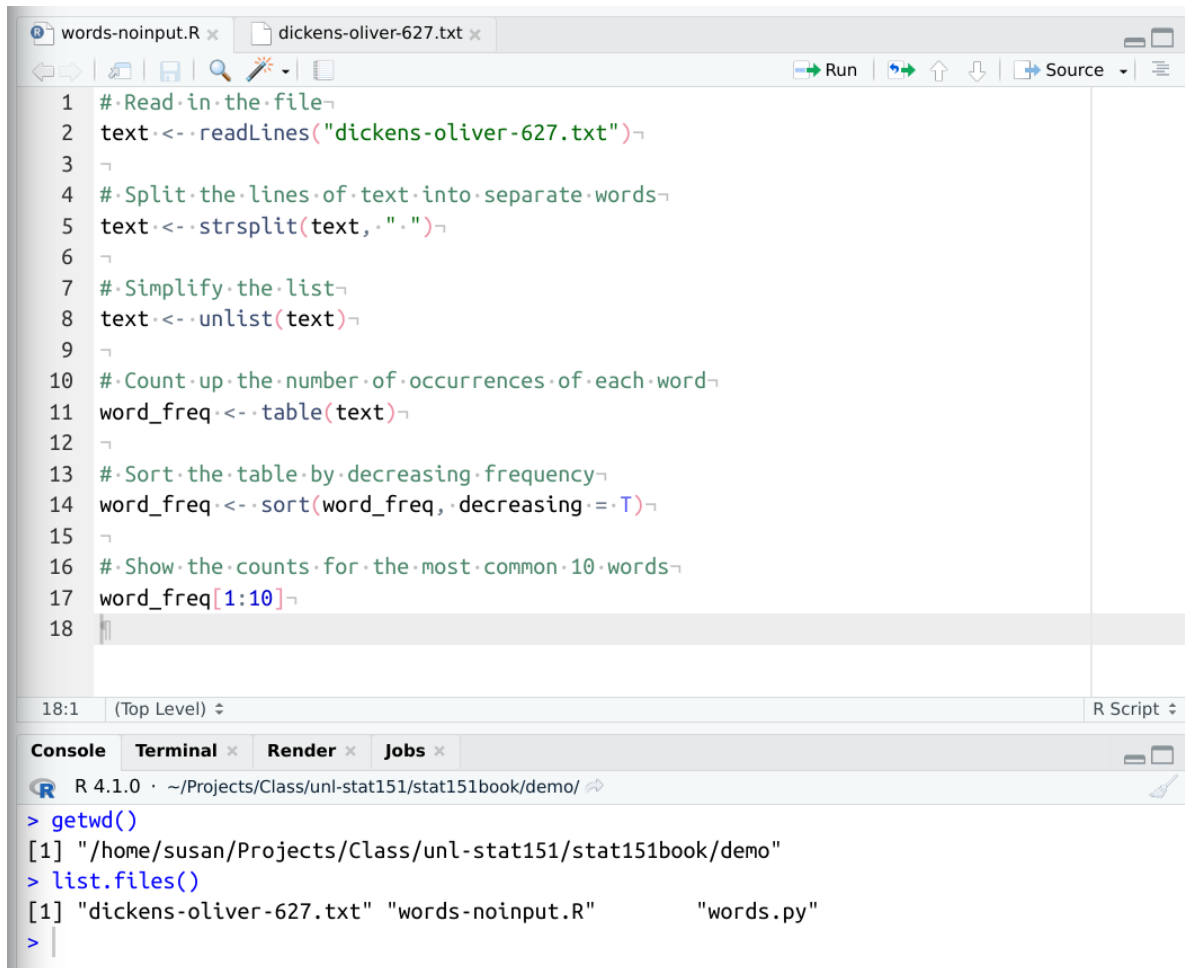
The image shows an R editor window with a script for analyzing word frequency in a text file. The script is as follows:

```
1 # Read in the file
2 text <- readLines("dickens-oliver-627.txt")
3
4 # Split the lines of text into separate words
5 text <- strsplit(text, " ")
6
7 # Simplify the list
8 text <- unlist(text)
9
10 # Count up the number of occurrences of each word
11 word_freq <- table(text)
12
13 # Sort the table by decreasing frequency
14 word_freq <- sort(word_freq, decreasing = T)
15
16 # Show the counts for the most common 10 words
17 word_freq[1:10]
18
```

Below the script editor is the R console window, which shows the output of the `getwd()` command:

```
> getwd()
[1] "/home/susan/Projects/Class/unl-stat151/stat151book/demo"
>
```

Figure 4.14: R editor window with relevant script, with R console shown below. My working directory is `/home/susan/Projects/Class/unl-stat151/stat151book/demo`; yours will be different.



The image shows an RStudio interface with two main panes. The top pane is the R Script editor, displaying a script named 'words-noinput.R'. The script contains 18 lines of R code that read a file, split it into words, and find the top 10 most frequent words. The bottom pane is the R Console, showing the output of the first two commands: `getwd()` and `list.files()`.

```
1 # Read in the file
2 text <- readLines("dickens-oliver-627.txt")
3
4 # Split the lines of text into separate words
5 text <- strsplit(text, ".")
6
7 # Simplify the list
8 text <- unlist(text)
9
10 # Count up the number of occurrences of each word
11 word_freq <- table(text)
12
13 # Sort the table by decreasing frequency
14 word_freq <- sort(word_freq, decreasing = T)
15
16 # Show the counts for the most common 10 words
17 word_freq[1:10]
18
```

18:1 (Top Level) R Script

R 4.1.0 · ~/Projects/Class/unl-stat151/stat151book/demo/

```
> getwd()
[1] "/home/susan/Projects/Class/unl-stat151/stat151book/demo"
> list.files()
[1] "dickens-oliver-627.txt" "words-noinput.R"      "words.py"
>
```

Figure 4.15: R editor window with relevant script, with R console shown below. dickens-oliver-627.txt is in the working directory, so we can proceed.

4.4.0.5 Example: Counting words in R in Interactive Mode

Using the file you created above, let's examine what each line does in interactive mode.

```
# Read in the file
text <- readLines("dickens-oliver-627.txt")
```

Select the above line and click the “Run” button in RStudio. Once you’ve done that, type in `text[1:5]` in the R console to see the first 5 lines of the file.

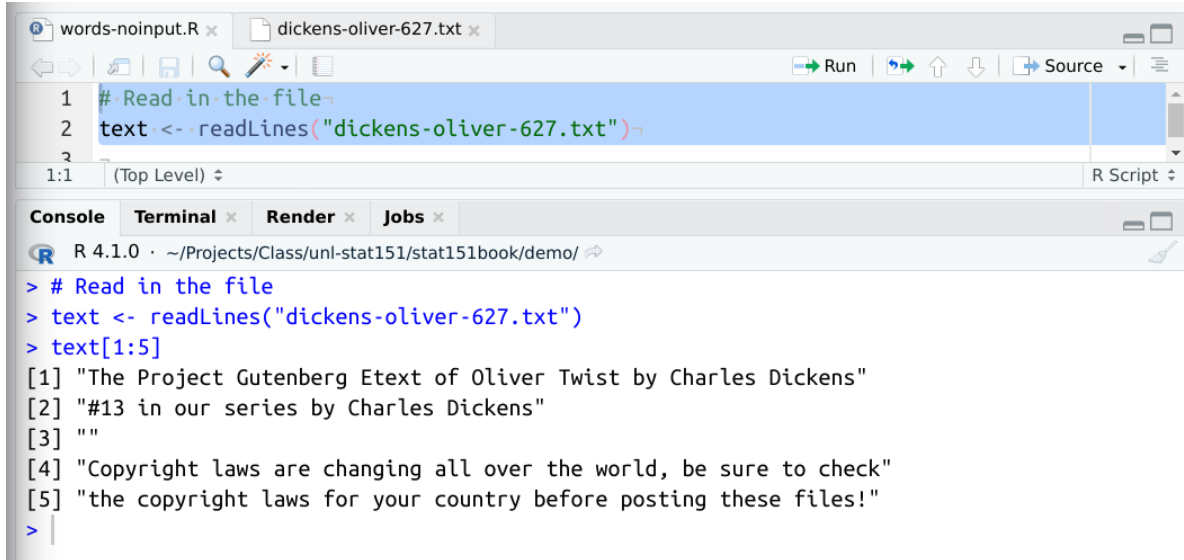


Figure 4.16: RStudio editor window with the first 2 lines of the `words-noinput.R` file selected. The screenshot also shows the console window after running the first 2 lines of the R file, with the `text[1:5]` command run interactively afterwards showing the first 5 lines of the text file we read in.

Run the next line of code using the run button (or click on the line of code and hit Ctrl/Cmd + Enter).

```
# Split the lines of text into separate words
text <- strsplit(text, " ")
```

Type in `text[[1]]` to see what the `text` object looks like now.

```
text[[1]]
```

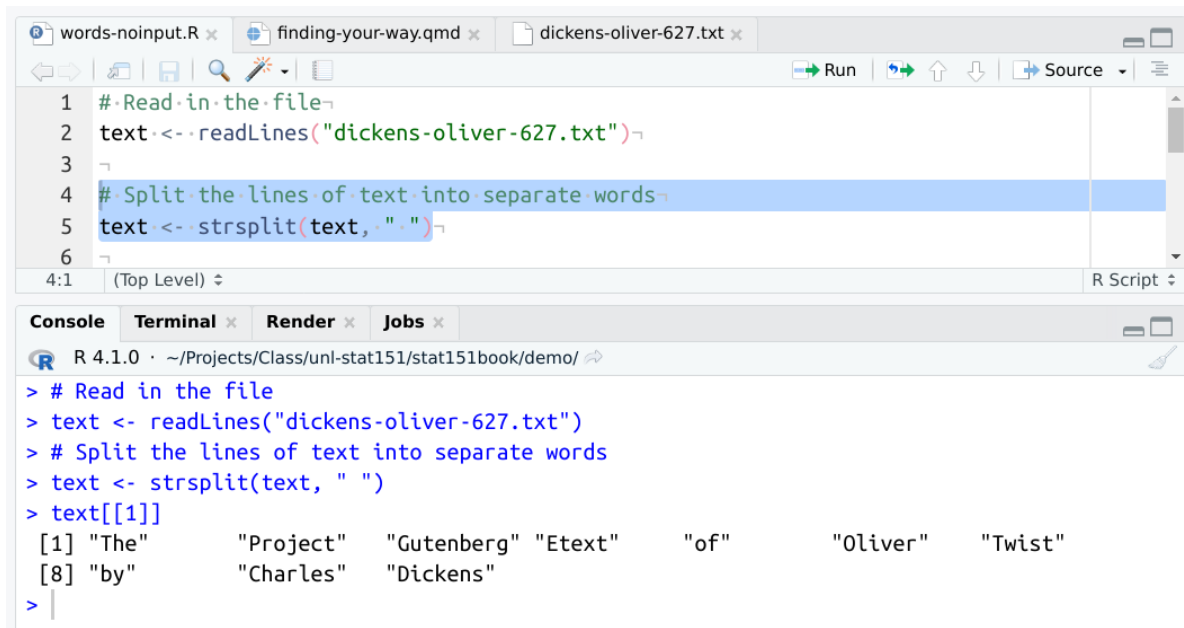


Figure 4.17: Screenshot of RStudio editor window with lines of code highlighted, plus RStudio console with the code as run and `text[[1]]` showing the first entry in the text object - a list of the separate words in the first line of the text file.

```
[1] "The"      "Project"  "Gutenberg" "Etext"    "of"       "Oliver"
[7] "Twist"    "by"       "Charles"   "Dickens"
```

```
# Simplify the list
text <- unlist(text)
text[[1]]
```

```
[1] "The"
```

```
text[1:20]
```

```
[1] "The"      "Project"  "Gutenberg" "Etext"    "of"       "Oliver"
[7] "Twist"    "by"       "Charles"   "Dickens"  "#13"      "in"
[13] "our"      "series"   "by"        "Charles"  "Dickens"  "Copyright"
[19] "laws"     "are"
```

Running `unlist` on `text` simplifies the object so that it is now a single **vector** of every word in the file, without regard for which line it appears on.

```
# Count up the number of occurrences of each word
word_freq <- table(text)
word_freq[1:5]
```

```
text
      _I_      --'      --by --kneel
4558      4        1        1        1
```

The next line assembles a table of frequency counts in `text`. There are 4558 spaces, 4 occurrences of the string `_I_`, and so on.

```
# Sort the table by decreasing frequency
word_freq <- sort(word_freq, decreasing = T)
word_freq[1:5]
```

```
text
the  and      to  of
8854 4902 4558 3767 3763
```

We can then sort `word_freq` so that the most frequent words are listed first. The final line just prints out the first 10 words instead of the first 5.

4.4.0.6 Example: Counting Words in R on the Command Line

Download the following file to your working directory: [words.R](#), or paste the following code into a new R script and save it as `words.R`.

```
# Take arguments from the command line
args <- commandArgs(TRUE)

# Read in the file
text <- readLines(args[1])

# Split the lines of text into separate words
text <- strsplit(text, " ")

# Simplify the list
text <- unlist(text)
```



```
# Count up the number of occurrences of each word
word_freq <- table(text)

# Sort the table by decreasing frequency
word_freq <- sort(word_freq, decreasing = T)

# Show the counts for the most common 10 words
word_freq[1:10]
```

In a terminal window opened at the location you saved the file (and the corresponding text file), enter the following: `Rscript words.R dickens-oliver-627.txt`.

Here, `Rscript` is the command that tells R to evaluate the file, `words.R` is the R code to run, and `dickens-oliver-627.txt` is an argument to your R script that tells R where to find the text file. This is similar to the python code, but instead, the user passes the file name in at the same time as the script instead of having to wait around a little bit.

4.4.1 Comparing Python and R

This is one good example of the difference in culture between python and R: python is a general-purpose programming language, where R is a domain specific programming language. In both languages, I've shown you how I would run the script by default first - in python, I would use a pre-built script to run things, and in R I would open things up in RStudio and source the script rather than running R from the command line.

This is a bit of a cultural difference – because python is a general purpose programming language, it is easy to use for a wide variety of tasks, and is a common choice for creating scripts that are used on the command line. R is a domain-specific language, so it is extremely easy to use R for data analysis, but that tends to take place (in my experience) in an interactive or script-development setting using RStudio. It is less natural to me to write an R script that takes input from the user on the command line, even though obviously R is completely capable of doing that task. More commonly, I will write an R script for my own use, and thus there is no need to make it easy to use on the command line, because I can just change it in interactive mode. Python scripts, on the other hand, may be written for a novice to use at the command line with no idea of how to write or modify python code. This is a subtle difference, and may not make a huge impression on you now, but it is something to keep in mind as you learn to write code in each language – the culture around python and the culture around R are slightly different, and this affects how each language is used in practice.

4.5 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`.

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

```
for?
```

(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 - usually, these pages will have examples. A similar set of pages exists for [R help on basic functions](#)

- [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.

5 Basic Data Types

5.1 Values and 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 both 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.

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

```
class(FALSE)
class(2L) # by default, R treats all numbers as numeric/decimal values.
          # The L indicates that we're talking about an integer.
class(2)
class("Hello, programmer!")
```

¹This 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.

```
[1] "logical"
[1] "integer"
[1] "numeric"
[1] "character"
```

```
type(False)
type(2)
type(3.1415)
type("This is python code")
```

```
<class 'bool'>
<class 'int'>
<class 'float'>
<class 'str'>
```

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.

5.2 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.

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.

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

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

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.

5.2.1 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?

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

```
Error: <text>:1:2: unexpected symbol
1: 1st_thing
   ^
```

```
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 not defined
```

Detailed traceback:

```
File "<string>", line 1, in <module>
```

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.

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.

[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.

5.3 Type Conversions

We talked about values and types above, but skipped over a few details because we didn't know enough about variables. It's now time to come back to those details.

What happens when we have an integer and a numeric type and we add them together? Hopefully, you don't have to think too hard about what the result of `2 + 3.5` is, but this is a bit more complicated for a computer for two reasons: storage, and arithmetic.

In days of yore, programmers had to deal with memory allocation - when declaring a variable, the programmer had to explicitly define what type the variable was. This tended to look something like the code chunk below:

```
int a = 1
double b = 3.14159
```

Typically, an integer would take up 32 bits of memory, and a double would take up 64 bits, so doubles used 2x the memory that integers did. Both R and python are **dynamically typed**, which means you don't have to deal with any of the trouble of declaring what your variables will hold - the computer automatically figures out how much memory to use when you run the code. So we can avoid the discussion of memory allocation and types because we're using higher-level languages that handle that stuff for us².

But the discussion of types isn't something we can completely avoid, because we still have to figure out what to do when we do operations on things of two different types - even if memory isn't a concern, we still have to figure out the arithmetic question.

So let's see what happens with a couple of examples, just to get a feel for **type conversion** (aka **type casting** or **type coercion**), which is the process of changing an expression from one data type to another.

```
mode(2L + 3.14159) # add 2 and pi
```

```
[1] "numeric"
```

```
mode(2L + TRUE) # add integer 2 and TRUE
```

```
[1] "numeric"
```

```
mode(TRUE + FALSE) # add TRUE and FALSE
```

```
[1] "numeric"
```

In R, all of the examples above are 'numeric' - basically, a catch-all class for things that are in some way, shape, or form numbers. Integers and decimal numbers are both numeric, but so are logicals (because they can be represented as 0 or 1).

```
type(2 + 3.14159)
```

```
<class 'float'>
```

```
type(2 + True)
```

```
<class 'int'>
```

²In some ways, this is like the difference between an automatic and a manual transmission - you have fewer things to worry about, but you also don't know what's going on under the hood nearly as well

```
type(True + False)
```

```
<class 'int'>
```

In python, by contrast, anything without a decimal point is converted into an integer - essentially, python tries to minimize memory used without losing data. So it will never convert a float into an integer implicitly - if you want Python to do that, you'll have to tell it to do so directly.

You may be asking yourself at this point why this matters, and that's a decent question. We will eventually be reading in data from spreadsheets and other similar tabular data, and types become *very* important at that point, because we'll have to know how R and python both handle type conversions.

In class activity

Do a bit of experimentation - what happens when you try to add a string and a number? Which types are automatically converted to other types? Fill in the following table in your notes:

Adding a ____ and a ____ produces a ____:

Logical	Integer	Decimal	String
Logical			
Integer			
Decimal			
String			

Above, we looked at automatic type conversions, but in many cases, we also may want to convert variables manually, specifying exactly what type we'd like them to be. A common application for this in data analysis is when there are "*" or "." or other indicators in an otherwise numeric column of a spreadsheet that indicate missing data: when this data is read in, the whole column is usually read in as character data. So we need to know how to tell R and python that we want our string to be treated as a number, or vice-versa.

In R, we can explicitly convert a variable's type using `as.XXX()` functions, where XXX is the type you want to convert to (`as.numeric`, `as.integer`, `as.logical`, `as.character`, etc.).

```
x <- 3
y <- "3.14159"
```



```
x + y
```

Error in x + y: non-numeric argument to binary operator

```
x + as.numeric(y)
```

```
[1] 6.14159
```

In python, the same basic idea holds true, but in python, we just use the variable type as a function: `int()`, `float()`, `str()`, and `bool()`.

```
x = 3
y = "3.14159"
x + y
```

Error in py_call_impl(callable, dots\$args, dots\$keywords): TypeError: unsupported operand type

Detailed traceback:

File "<string>", line 1, in <module>

```
x + float(y)
```

```
6.14159
```

5.4 Operators and Functions

In addition to variables, **functions** are extremely important in programming.

Let's first start with a special class of functions called operators. You're probably familiar with operators as in arithmetic expressions: `+`, `-`, `/`, `*`, and so on.

Here are a few of the most important ones:

Operation	R symbol	Python symbol
Addition	+	+
Subtraction	-	-
Multiplication	*	*

Operation	R symbol	Python symbol
Division	/	/
Integer Division	%%	//
Modular Division	%%	%
Exponentiation	^	**

Note that integer division is the whole number answer to A/B , and modular division is the fractional remainder when A/B . So `14 %% 3` in R would be 4, and `14 %% 3` in R would be 2.

```
14 %% 3
```

```
[1] 4
```

```
14 % 3
```

```
[1] 2
```

```
14 // 3
```

```
4
```

```
14 % 3
```

```
2
```

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

5.4.1 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.

```
(1+1)^(5-2) # 2 ^ 3 = 8
```

```
[1] 8
```

```
1 + 2^3 * 4 # 1 + (8 * 4)
```

```
[1] 33
```

```
3*1^3 # 3 * 1
```

```
[1] 3
```

```
(1+1)**(5-2)
```

```
8
```

```
1 + 2**3*4
```

```
33
```

```
3*1**3
```

```
3
```

5.4.2 String Operations in Python

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.

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

```
"first " + "second"
```

```
'first second'
```

```
"hello " * 3
```

```
'hello hello hello '
```

5.4.3 Functions

Functions are sets of instructions that take **arguments** and **return** values. Strictly speaking, 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.

It may be helpful at this point to print out the [R reference card](#)³ and the [Python reference card](#)⁴. These cheat sheets contain useful functions for a variety of tasks in each language.

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.

In class activity

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?

³From <https://cran.r-project.org/doc/contrib/Short-refcard.pdf>

⁴From http://sixthresearcher.com/wp-content/uploads/2016/12/Python3_reference_cheat_sheet.pdf

6 Data and Control Structures

This chapter introduces some of the most important tools for working with data: vectors, matrices, loops, and if statements. It would be nice to gradually introduce each one of these topics separately, but they tend to go together, especially when you're talking about programming in the context of data processing.

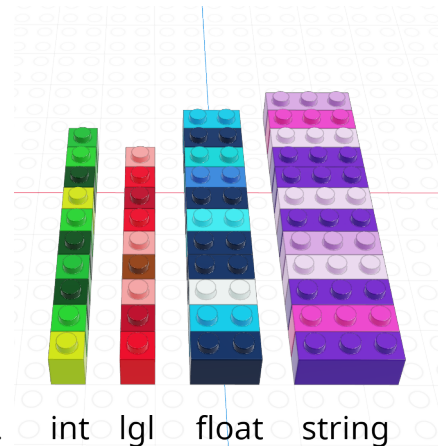
In the previous chapter, we discussed 4 different data types: strings/characters, numeric/double/floats, integers, and logical/booleans. As you might imagine, things are about to get more complicated.

Data **structures** are more complicated arrangements of information.

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

6.1 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 want¹:

6.1.1 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
Count elements as 1, 2, 3, 4, ..., N	Count elements as 0, 1, 2, 3, , ..., N-1



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

Creating vectors in R

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

# Access individual entries
digits_pi[1]
```

¹Throughout 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!

```
[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
```

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

Creating vectors in python

```
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]
```

²A 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.

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]
```

```
# Print out the vector
```

Error in py_call_impl(callable, dots\$args, dots\$keywords): IndexError: index 11 is out of bounds

Detailed traceback:

File "<string>", line 1, in <module>

```
print(digits_pi)
```

```
[3 1 4 1 5 9 2 6 5 3 5]
```

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

```
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"
```

If you're curious about any of these cats, see the footnotes³.

6.1.2 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**.

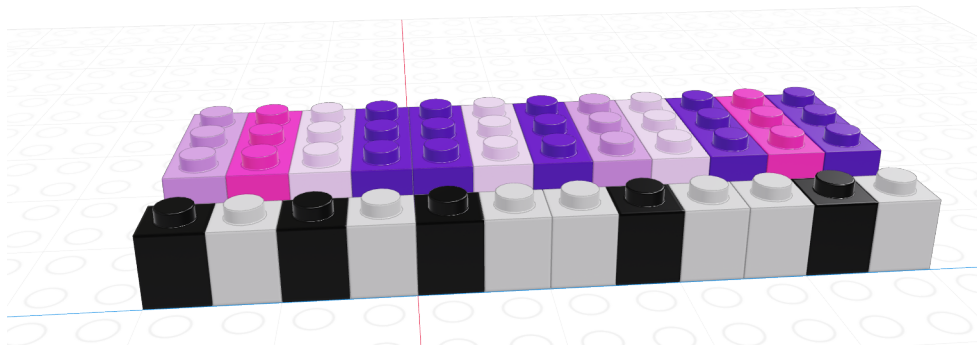


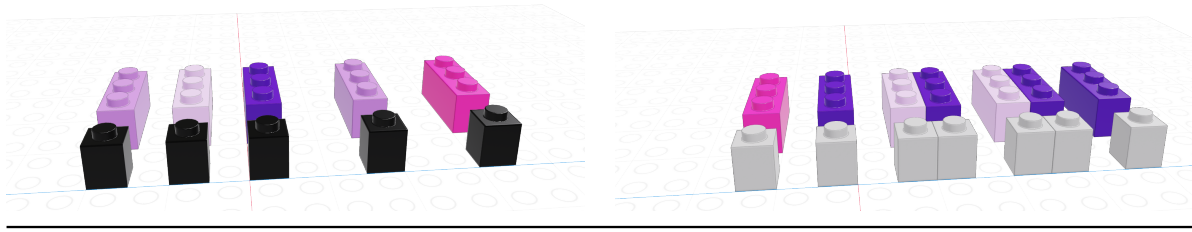
Figure 6.1: lego vectors - a pink/purple hued set of 1x3 bricks representing the data and a corresponding set of 1x1 grey and black bricks representing the logical index vector of the same length

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.

³- Grumpy cat: <https://www.grumpycats.com/> - Garfield: <https://www.garfield.com/> - Nyan cat: https://en.wikipedia.org/wiki/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.

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.

```
# 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"
```

```
import numpy as np;

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

# 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')
```

6.1.3 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?

```
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
```

```
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 numerics
```

```
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.

6.2 Matrices

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

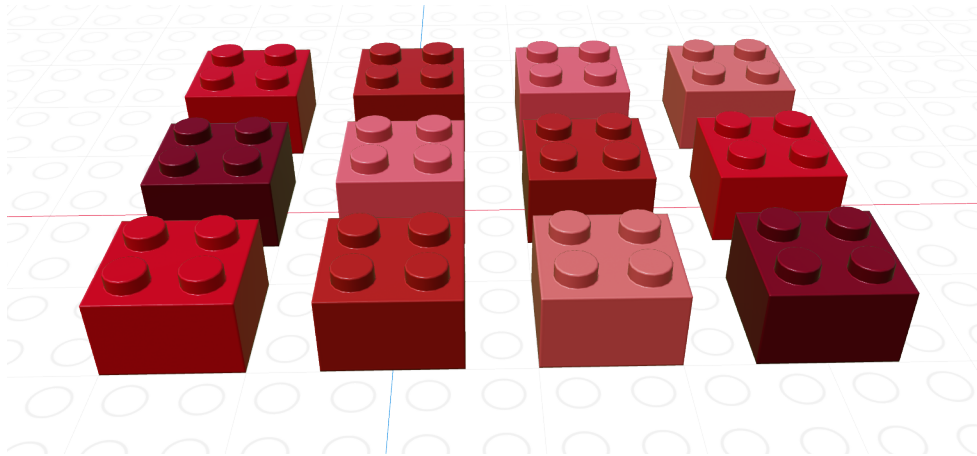


Figure 6.2: lego depiction of a 3-row, 4-column matrix of 2x2 red-colored blocks

```
# 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
```

```
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
```

```
import numpy as np
# Minimal matrix in python
np.mat([[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([[ 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?

6.2.1 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.

```
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
```

```
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]])
```

7 Data Structures

data frames in R and Pandas

8 Reading in Data

External data files - csv, excel, etc.

References

Severance, Dr Charles Russell. 2016. *Python for Everybody: Exploring Data in Python 3*. Edited by Sue Blumenberg and Elliott Hauser. Ann Arbor, MI: CreateSpace Independent Publishing Platform. <https://www.py4e.com/html3/>.