An Introduction to R and Python For Data Analysis: A Side By Side Approach

Taylor R. Brown

Contents

Welcome **9**

A Sample Course 9

Become a Contributor 9

License(s) 9

Preface **11**

About this book 11

Conventions 12

Installing the Required Software 13

1 Getting Started **15**

1.1 Hello World in R 15

1.2 Hello World in Python 18

1.3 Getting Help 21

1.4 Exercises 23

2 Data Types **25**

2.1 Basic Types 25

2.2 R vectors versus Numpy `arrays` (and Pandas `Series`) 29

2.3 Numpy's `ndarrays` versus R's matrices and arrays 40

2.4 R's lists versus Python's lists and dictionaries 41

2.5 User-defined functions in R and Python 43
2.6 Categorical Data 63
2.7 Data Frames 67
2.8 Exercises 74

3 Odds and Ends 79

3.1 Input and Output 79
3.2 Using Third-Party Code 80
3.3 Control Flow 87
3.4 Reshaping and Combining Data Sets 94
3.5 Visualization 106
3.6 Exercises 116

4 An Introduction to Object-Oriented Programming 117

4.1 OOP In Python 119
4.2 OOP In R 127
4.3 Exercises 141

5 Functional Programming 145

5.1 Functions as Function Inputs in R 147
5.2 Another Example in R 157
5.3 Functions as Function Inputs in Base Python 159
5.4 Functions as Function Inputs in Numpy 161
5.5 Functional Methods in Pandas 162
5.6 Functions as Function Inputs (miscellany) 165
5.7 Functions as Function Outputs in R 165
5.8 Functions as Function Outputs in Python 169

List of Tables

List of Figures

1.1 RStudio 16
1.2 Anaconda Navigator 19
1.3 Spyder 19

Welcome

A Sample Course

Please email me for access to a private Github repository with course materials such as a syllabus, course schedule, and assignments.

Assignments are written with automatic grading in mind. R scripts are graded with the `gradeR` package, and Python scripts with the `Otter Grader` package (TODO cite). If you want to generate your own autograding files, you may use the tool here (TODO).

Become a Contributor

Spot a typo, or have a suggestion? Feel free to post an **issue here**. You may also submit pull requests through Github. I'll be happy to take a look.

License(s)

Preface

About this book

This book is written to be used in a one-semester statistical computing class that teaches both R and Python to graduate students in a statistics or data science department. This book is written for students that do not necessarily possess any previous familiarity with writing code.

- If you are using them for analyzing data, R and Python do a lot of the same stuff in pretty similar ways, so it does not always make sense to teach one language after the other. Imagine learning about vectorization, say, in R, and then several weeks later learning about the same concept in Python. You might end up spending a lot of time refreshing your memory before getting started for the second time. The side-by-side approach ought to help reinforce shared concepts. In my opinion, it also helps to highlight key differences.

- This text does not describe statistical modeling techniques in detail, although many exercises will be motivated by different statistical techniques. Rather, it teaches the basics of data "cleaning", "munging" and manipulation.

- This book is written for aspiring data scientists, not necessarily computer scientists. Why do I draw the distinction? When discussing different types, for example, I do not discuss data structures in any depth. Rather, I discuss examples of applications where different types would be most useful.

- It is meant to be read on a computer with an internet connection. There are plenty of hyperlinks to click, and some interactive visualizations. These don't work if you have a text copy.

- This book does not attempt to be an authoritative reference. This is my attempt at balancing depth and breadth for a one-semester course. Plenty of discovery will be left to the reader. In my opinion, it is not possible to learn everything about R and Python for doing data science in one semester. The hyperlinks are there to support self-reliance and self-sufficiency. Also, I imagine it being used as a reference. It's more important that students do the exercises on their own. Associating exercises with a (sub-)section in the textbook will give them support when they need it.

- Chapters may be read out of order, but (sub-)sections within a chapter are carefully ordered. Say, for instance, a topic in R is discussed first. If the Python discussion comes second, that discussion will reference and make comparisons with details mentioned in the R section. So read sections and subsections in order.

Conventions

Sometimes R and Python code look very similar. At times they even look identical. This is why we usually separate R and Python code into separate sections.

However, whenever it is necessary to prevent confusion, I will remind you what language is being used in comments (more about comments in 1.2).

```
# in python
print('hello world')
## hello world
```

```
# in R
print('hello world')
## [1] "hello world"
```

Installing the Required Software

To get started, you must install both R and Python. The installation process depends on what kind of machine you have (e.g. what type of operating system your machine is running, is your processor 32 or 64 bit, etc.).

I provide some helpful links below. However, there are many alternatives available to you for how to install, and different options may be preferable to different students. Moreover, these options are likely to change over time.

If you prefer to take a different route, or you already have a solution to either of these, feel free to keep using it. The code should still work as long as you're using relatively recent versions of each. For Python, you need a version that is `>=3.6`, and for R, it nees to be `>=4.0.0`.

Installing R (and RStudio)

It is recommended that you install R and *RStudio Desktop*. *RStudio Desktop* is a graphical user interface with many tools that making writing R easier and more fun.

Install R from the Comprehensive R Archive Network (CRAN). You can access instructions for your specific machine by clicking here.

You can get RStudio Desktop directly from the company's website.

Installing Python by Installing Anaconda

It is recommended that you install *Anaconda*, which is a package manager, environment manager, and Python distribution with many third party open source packages. It provides a graphical user interface for us, too, just as RStudio does. You can access instructions for your specific machine and OS by clicking here.

Chapter 1

Getting Started

Now that you have both R and Python installed, we can get started by taking a tour of our two different development environments `RStudio` and `Spyder`.

In addition, I will also discuss a few topics superficially, so that we can get our feet wet:

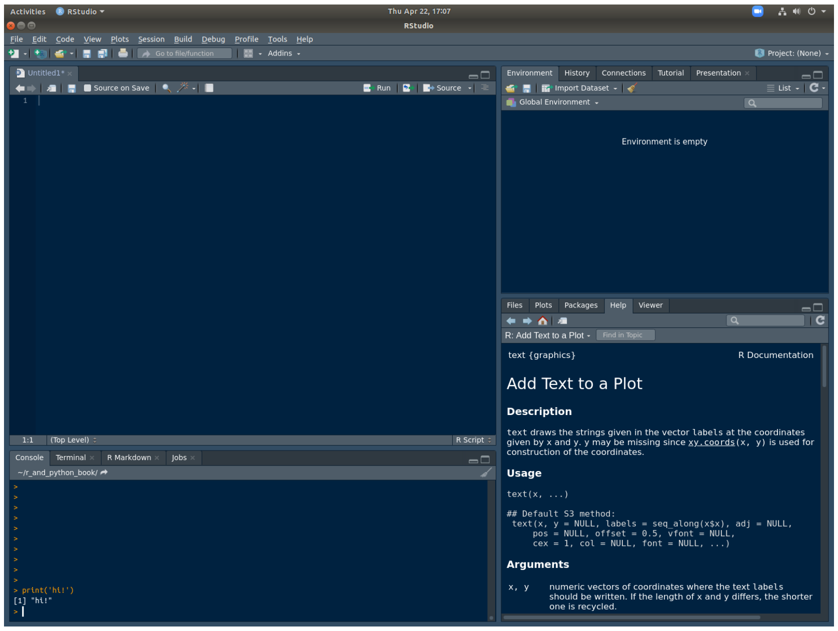
- printing,
- creating variables, and
- calling functions.

1.1 Hello World in R

Go ahead and open up `RStudio`. It should look something like this

I changed my "Editor Theme" from the default to "Cobalt" because it's easier on my eyes. If you are opening `RStudio` for the first time, you probably see a lot more white. You can play around with the theme, if you wish, after going to `Tools -> Global Options -> Appearance`.

The **console**, which is located by default on the lower left panel, is the place that all of your code gets run. For short one-liners, you can type code directly into the console. Try typing the following code in there. Here we are making use of the `print()` function.



```
>
>
>
>
>
>
>
>
>
>
> print('hi!')
[1] "hi!"
>
```

text {graphics}   R Documentation

Add Text to a Plot

Description

text draws the strings given in the vector labels at the coordinates given by x and y. y may be missing since xy.coords(x, y) is used for construction of the coordinates.

Usage

```
text(x, ...)

## Default S3 method:
 text(x, y = NULL, labels = seq_along(x$x), adj = NULL,
     pos = NULL, offset = 0.5, vfont = NULL,
     cex = 1, col = NULL, font = NULL, ...)
```

Arguments

x, y   numeric vectors of coordinates where the text labels should be written. If the length of x and y differs, the shorter one is recycled.

Figure 1.1: RStudio

In R, functions are "first-class objects," which means can refer to the name of a function without asking it to do anything. However, when we *do* want to use it, we put parentheses after the name. This is called **calling** the function or **invoking** the function. If a function call takes any **arguments** (aka inputs), then the programmer supplies them between the two parentheses. A function may **return** values to be subsequently used, or it may just produce a "side-effect" such as printing some text, displaying a chart, or read/writing information to an external data source.

```r
print('hello R world')
## [1] "hello R world"
```

During the semester, we will write more complicated code. Complicated code is usually written incrementally and stored in a text file called a **script**. Click `File -> New File -> R Script` to create a new script. It should appear at the top left of the `RStudio` window (see Figure 1.1) . After that, copy and paste the following code into your script window.

```r
print('hello world')
print("this program")
print('is not incredibly interesting')
print('but it would be a pain')
print('to type it all directly into the console')
myName <- "Taylor"
print(myName)
```

This script will run five print statements, and then create a variable called `myName`. The print statements are of no use to the computer and will not affect how the program runs. They just display messages to the human running the code.

The variable created on the last line is more important because it is used by the computer, and so it can affect how the program runs. The operator `<-` is the **assignment operator**. It takes the character constant `"Taylor"`, which is on the right, and stores it under the name `myName`. If we added lines to this program, we could refer to the variable `myName` in subsequent calculations.

Save this file wherever you want on your hard drive. Call it `awesomeScript.R`. Personally, I saved it to my desktop.

After we have a saved script, we can run it by sending all the lines of code over to the console. One way to do that is by clicking the `Source` button at the top right of the script window (see Figure 1.1).

Another way is that we can use R's `source()` function. We can run the following code in the console.

```r
# Anything coming after the pound/hash-tag symbol
# is a comment to the human programmer.
# These lines are ignored by R
setwd("/home/taylor/Desktop/")
source("awesomeScript.R")
```

The first line changes the **working directory** to `Desktop/`. You, dear reader, should change this line by replacing `Desktop/` to whichever folder you chose to save `awesomeScript.R` in. If you would like to find out what your working directory is currently set to, you can use `getwd()`.

Every computer has a different folder/directory structure–that is why it is highly recommended you refer to file locations as seldom as possible in your scripts. This makes your code more *portable*. When you send your file to someone else (e.g. your instructor or your boss), she will have to remove or change every mention of any directory. This is because those directories (probably) won't exist on her machine.

The second line calls `source()`. This function finds the script file and executes all the commands found in that file sequentially.

A third way is to tell R to run `awesomeScript.R` from the command line. We will describe this approach in more detail in chapter TODO.

1.2 Hello World in Python

This section will assume you already have Python installed.

First, start by opening *Anaconda Navigator*. It should look something like this:

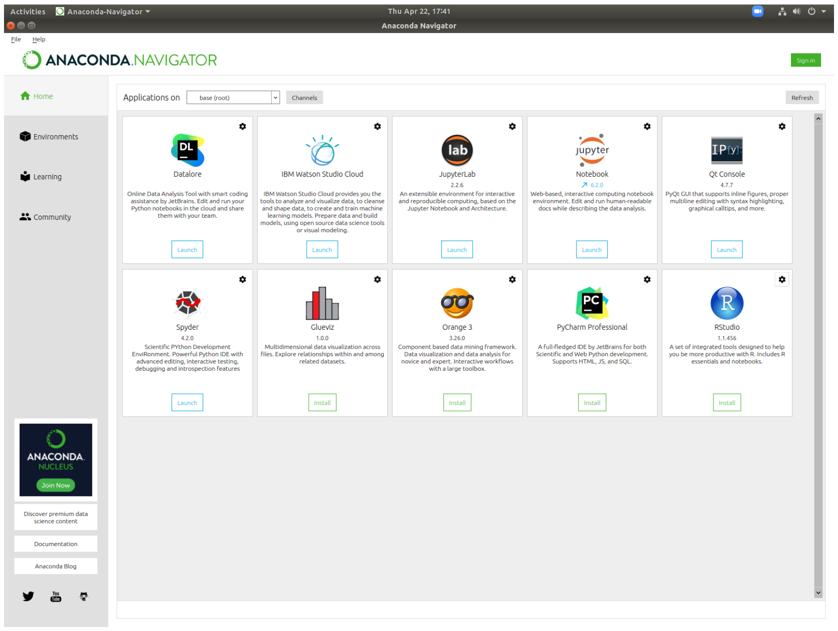
We will exclusively use *Spyder* for this course. Open that up now. It should look something like this:

It looks a lot like `RStudio`, right?! The script window is still on the left hand side, but it takes up the whole height of the window this time. However, you will notice that the console window has moved. It's over on the bottom right now.

Again, you might notice a lot more white when you open this for the first time. Just like last time, I changed my color scheme. You can change yours by going to `Tools -> Preferences` and then exploring the options available under the `Appearances` tab.

Try typing the following line of code into the console.

```
# this looks like R code but it's Python code!
print("hello Python world")
## hello Python world
```



Datalore
Online Data Analysis Tool with smart coding assistance by JetBrains. Edit and run your Python notebooks in the cloud and share them with your team.
Launch

IBM Watson Studio Cloud
IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.
Launch

JupyterLab
2.2.6
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch

Notebook
↗ 6.2.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch

Qt Console
4.7.7
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
Launch

Spyder
4.2.0
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch

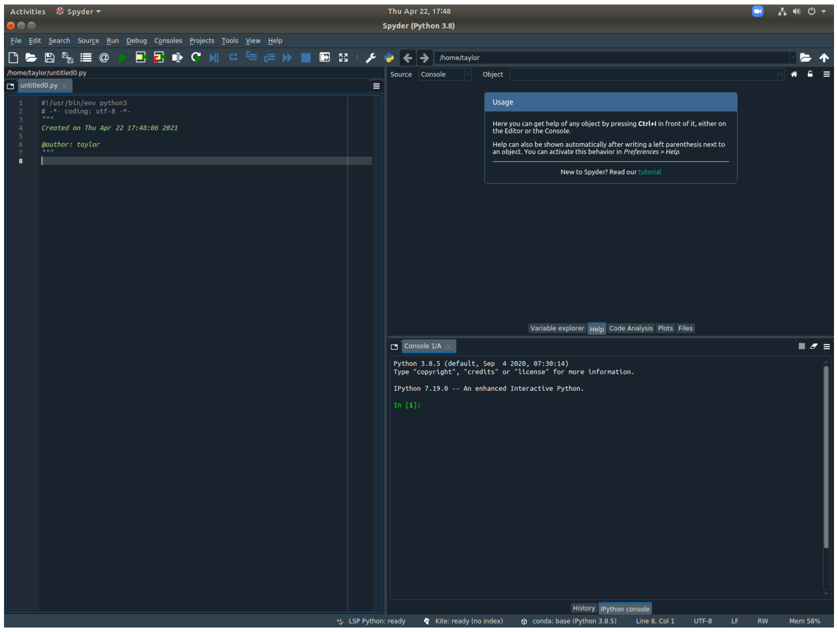
Glueviz
1.0.0
Multidimensional data visualization across files. Explore relationships within and among related datasets.
Install

Orange 3
3.26.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.
Install

PyCharm Professional
A full-fledged IDE by JetBrains for both Scientific and Web Python development. Supports HTML, JS, and SQL.
Install

RStudio
1.1.456
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.
Install

Figure 1.2: Anaconda Navigator



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 22 17:48:06 2021

@author: taylor
"""

```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help.*

New to Spyder? Read our tutorial

```
Python 3.8.5 (default, Sep  4 2020, 07:30:14)
Type "copyright", "credits" or "license" for more information.

IPython 7.19.0 -- An enhanced Interactive Python.

In [1]:
```

LSP Python: ready   Kite: ready (no index)   conda: base (Python 3.8.5)   Line 8, Col 1   UTF-8   LF   RW   Mem 58%

Figure 1.3: Spyder

Already we have many similarities between our two languages. Both R and Python have a `print()` function, and they both use the same symbol to start a comment: `#`. Finally, they both define character/string constants with quotation marks In both languages, you can use either single or double quotes.

We will also show below that both languages share the same three ways to run scripts. Nice!

Let's try writing our first Python script. R scripts end in `.r` or `.R`, while Python scripts end in `.py`. Call this file `awesomeScript.py`.

```
# save this as awesomeScript.py
print('hello world')
print("this program")
print('is pretty similar to the last program')
print('it is not incredibly interesting, either')
my_name = "Taylor"
print(myName)
```

Notice that the assignment operator is different in Python. It's an `=`[1].

Just like `RStudio`, `Spyder` has a button that runs the entire script from start to finish. It's the green triangle button (see Figure 1.3).

You can also write code to run `awesomeScript.py`. There are a few ways to do this, but here's the easiest.

```
import os
os.chdir('/home/taylor/Desktop')
runfile("awesomeScript.py")
```

This is also pretty similar to the R code from before. `os.chdir()` sets our working directory to the `Desktop`. Then `runfile()` runs all of the lines in our program, sequentially, from start to finish.

The first line is new, though. We did not mention anything like this in R, yet. We will talk more about `import`ing modules in section 3.2.4. Suffice it to say

[1]You can use this symbol in R, too, but it is less common.

that we imported the `os` module to make the `chdir()` function available to us.

Third, we can tell Python to run `awesomeScript.py` from the command line. We will describe this approach in more detail in chapter TODO.

1.3 Getting Help

1.3.1 Reading Documentation

Programming is not about memorization. Nobody can memorize, for example, every function and all of its arguments. Nobody can do that. Not even close. So what do programmers do when they get stuck? The primary way is to read the documentation.

1.3.1.1 In R

Getting help in R is easy. If you want to know more about a function, type into the console the name of the function with a leading question mark. For example, `?print` or `?setwd`. You can also use `help()` and `help.search()` to find out more about functions (e.g. `help(print)`). Sometimes you will need to put quotation marks around the name of the function (e.g. `?":"`).

This will not open a separate web browser window, which is very convenient. If you are using `RStudio`, you have some extra benefits. Everything will look very pretty, and you can search through the text by typing phrases into the search bar in the "Help" window.

1.3.1.2 In Python

In Python, the question mark comes *after* the name of the function[2] (e.g. `print?`), and you can use `help(print)` just as in R.

[2]If you did not install `Anaconda`, then this may not work for you because this is an IPython (https://ipython.org) feature.

In Spyder, if you want the documentation to appear in the Help window (it looks prettier), then you can type the name of the function, and then `Ctrl-i` (`Cmd-i` on a mac keyboard).

1.3.2 Understanding File Paths

File paths look different on different operating systems. Mac and Linux machines tend to have forward slashes (i.e. `/`), while Windows machines tend to use backslashes (i.e. `\`).

Depending on what kind of operating system is running your code, you will need to change the file paths. It is important for everyone writing R and Python code to understand how things work on both types of machines–just because you're writing code on a Windows machine doesn't mean that it won't be run on a Mac, or vice versa.

1.3.2.1 *nix Machines

The directory repeatedly mentioned in the code above was `/home/taylor/Desktop`. This is a directory on my machine which is running Ubuntu Linux. The leading forward slash is the *root directory*. Inside that is the directory `home/`, and inside that is `taylor/`, and inside that is `Desktop/`. If you are running MacOS, these file paths will look very similar. The folder `home/` will most likely be replaced with `Users/`.

1.3.2.2 Windows Machines

On Windows, things are a bit different. For one, a full path starts with a drive (e.g. `C:`). Second, there are backslashes (not forward slashes) to separate directory names (e.g `C:\Users\taylor\Desktop`).

Unfortunately, backslashes are a special character in both R and Python. Whenever you type a `\`, it will change the meaning of whatever comes after it. This is what is known as an **escape character**.

In both R and Python, the backslash character is used to start an "escape" sequence. You can see some examples in R by clicking here, and some examples in Python by clicking here. In Python it may also be used to allow long lines of code to take up more than one line in a text file.

The recommended way of handling this is to just use forward slashes instead. For example, if you are running Windows, `C:/Users/taylor/Desktop/myScript.R` will work in R, and `C:/Users/taylor/Desktop/myScript.py` will work in Python.

You may also use "raw string constants" (e.g. `r'C:\Users\taylor\Desktop\my_file.txt'`), or you can "escape" the backslashes by replacing each single backslash with a double backslash.

1.4 Exercises

1.4.1 Discussion

1. What are some similarities between R and Python? Which of these are you most thankful for? Which of them are you most concerned about?

2. What are some major differences between R and Python? What do you think about these differences?

1.4.2 Easy

1. Create an R script called `my_script.R`. Inside that file,

   - create a variable called `myName` and assign your name to it.
   - create another variable called `myFavNum` and assign to it your favorite number.
   - create a variable called `rMessage` and assign to it the following phrase `"camelCase is cool for naming variables in R"`

2. Create a Python script called `my_script.py`. Inside that file

   - create a variable called `my_name` and assign your name to it.
   - create another variable called `my_fav_num` and assign to it your favorite number.

- create a variable called `py_message` and assign to it the following phrase `"snake_case is cool for naming variables in Python"`[3]

[3]More suggestions on writing stylish Python code can be found here. Regarding my suggesting camel case, other style guides disagree with this. Don't worry about this too much, though–after all "a foolish consistency is the hobgoblin of little minds."

Chapter 2

Data Types

In every programming language, data is stored in different ways. Writing a program that manipulates data requires understanding all of the choices. That is why we must be concerned with the different **types** of data in our R and Python programs. Different types are suitable for different purposes.

There are similarities between Python's and R's type systems. However, there are may differences as well. Be prepared for these differences. There are many more of them in this chapter than there were in the previous chapter!

If you're ever unsure what type a variable has, use `type()` (in Python) or `typeof()` (in R) to query it.

2.1 Basic Types

Storing an individual piece of information is simple in both languages. However, while Python has scalar types, R's situation is a little more complicated.

2.1.1 In Python

In Python, the simplest types we frequently use are `str` (short for string), `int` (short for integer), `float` (short for floating point) and `bool` (short for Boolean). This list is not exhaustive, but these are a good collection to start thinking about. For a complete list of built-in types in Python, click here.

```
print(type('a'), type(1), type(1.3))
## <class 'str'> <class 'int'> <class 'float'>
```

Strings are useful for processing text data such as names of people/places/things and messages (e.g. texts, tweets and emails). If you are dealing with numbers, you need floating points if you have a number that might have a fractional part after its decimal; otherwise you'll need an integer. Booleans are useful for situations where you need to record whether something is true or false. They are also important to understand for control-flow in section 3.3.

In the next section we will discuss the Numpy library. This library has a broader collection of basic types.

2.1.1.1 Type Conversions in Python

We will often have to convert between types in a Python program. This is called **type conversion**, and it can be either implicitly or explicitly done. For example, `int`s are often implicitly converted to `float`s, so that arithmetic operations work.

```
my_int = 1
my_float = 3.2
my_sum = my_int + my_float
print("my_int's type", type(my_int))
## my_int's type <class 'int'>
print("my_float's type", type(my_float))
## my_float's type <class 'float'>
print(my_sum)
## 4.2
print("my_sum's type", type(my_sum))
## my_sum's type <class 'float'>
```

You might be disappointed if you always count on this behavior, though.

```
3.2 + "3.2"
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: unsupporte
```

Explicit conversions occur when we as programmers are explicitly asking Python to perform the conversion. You will do this with the functions `int()`, `str()`, `float()`, `bool()`, and the like.

```
my_date = "5/2/2021"
month_day_year = my_date.split('/')
my_year = int(month_day_year[-1])
print('my_year is equal to ', my_year, 'and its type is ', type(my_year))
## my_year is equal to  2021 and its type is  <class 'int'>
```

2.1.2 In R

In R, the names of these basic types are only slightly different. They are `logical` (instead of `bool`), `integer` (instead of `int`), `double` (instead of `float`)[1], and `character` (instead of `str`). There is also `complex` and `raw`, but we will use these less often in this textbook.

```
# cat() is kind of like print()
cat(typeof('a'), typeof(1), typeof(1.3))
## character double double
```

In this case R automatically upgraded `1` to a double. If you wanted to force it to be an integer, you can add a capital "L" to the end of the number.

```
# cat() is kind of like print()
cat(typeof('a'), typeof(1L), typeof(1.3))
## character integer double
```

[1]"double" is short for "double precision floating point." In the programming language `C`, which is what both R and Python are written in, the programmer has more control on how many decimal points of precision he wants.

2.1.2.1 Type Conversions in R

You can explicitly and implicitly convert types in R just as you did in Python. Implicit conversion looks like this.

```
myInt = 1
myDouble = 3.2
mySum = myInt + myDouble
print(paste0("my_int's type is ", typeof(myInt)))
## [1] "my_int's type is double"
print(paste0("my_float's type is ", typeof(myDouble)))
## [1] "my_float's type is double"
print(mySum)
## [1] 4.2
print(paste0("my_sum's type is ", typeof(mySum)))
## [1] "my_sum's type is double"
```

Explicit conversion can be achieved with functions such as `as.integer`, `as.logical`, `as.double`, etc.

```
print(typeof(1))
## [1] "double"
print(typeof(as.logical(1)))
## [1] "logical"
```

2.1.2.2 What was the weird thing about R you mentioned?

The basic types of R are a little different than the basic types of Python. R uses the same type to store many elements. It does not have any scalar type. On the other hand, Python has base types for individual elements, and it uses separate types as containers for storing many elements. In R, if you are looking at single number or character string, it's actually a length 1 `vector`. More information about `vector`s can be found in section 2.2.

TODO constants versus literals?

2.1.3 Exercises

2.1.3.1 Easy

All answers to questions related to R should be written in a file named `easy_data_types_exercises.R`. All answers to questions related to Python should be written in a file named `easy_data_types_exercises.py`.

1. Which Python type is ideal for each piece of data? Assign your answers to a `list` of `string`s called `question_one`.

   - An individual's IP address
   - whether or not an individual attended a study
   - the number of seeds found in a plant
   - the amount of time it takes for a car to race around a track

2. Answer the same question above, but use R types? Assign your answers to a `character vector` of length four called `questionOne`.

   - An individual's IP address
   - whether or not an individual attended a study
   - the number of seeds found in a plant
   - the amount of time it takes for a car to race around a track

3. "The only numbers that can be represented exactly in R's numeric type are integers and fractions whose denominator is a power of 2." Provide ten examples of numbers that have nonzero decimal components and are not exactly equal represented by R's `numeric` type. Store them all in a vector called `notExactFloats`.

2.2 R vectors versus Numpy `arrays` (and Pandas `Series`)

This section is for describing the data types that let us store collections of elements that all **share the same type**. Data is very commonly stored in this fashion, so this section is quite important. Once we have one of these objects in a program, we will be interested in learning how to extract different subsets of elements, and how vectorization works.

2.2.1 Overview of R

I mentioned earlier that R does not have scalar types–it just has **vectors**. So, whether you want to store one number, or many numbers, you will need a `vector`.

How do we create one of these? There are many ways. One common is to read in elements from an external data set, perhaps extracting them from a column of a `data.frame`, but we will describe those in section 2.7. Here are some examples of generating `vectors` from code instead of grabbing them from an external data set after it is read in.

```r
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
seq(1,10,2)
## [1] 1 3 5 7 9
rep(2,5)
## [1] 2 2 2 2 2
c("5/2/2021", "5/3/2021", "5/4/2021")
## [1] "5/2/2021" "5/3/2021" "5/4/2021"
rnorm(10)
##  [1] -1.8072  0.1382 -0.3399 -1.5573 -0.2622  0.1791
##  [7]  0.8533  1.0061  1.0880 -0.2615
```

`c` is short for "combine". `seq` and `rep` are short for "sequence" and "replicate", respectively. `rnorm` samples normal (or Gaussian) random variables. There is plenty more to learn about these functions, so I encourage you to take a look at their documentation.

2.2.2 Overview of Python

If you want to store many elements of the same type (and size) in Python, you will need a Numpy `array`. Numpy is a highly-regarded third party library (Harris et al., 2020) for Python.

There are five ways to create numpy arrays (source). Here are some examples that complement the examples from above.

```
import numpy as np
np.array([1,2,3])
## array([1, 2, 3])
np.arange(1,12,2)
## array([ 1,  3,  5,  7,  9, 11])
np.random.normal(size=3)
## array([ 0.64610205, -0.17502578,  0.16411466])
```

Another choice in Python is to use a `Series` object from the `Pandas` library. The benefit of these is that they play nicely with Pandas data frames (more information about Pandas data frames can be found in 2.7.2), and that they have more flexibility with accessing elements by name (see here for more information).

```
import pandas as pd
first = pd.Series([2, 4, 6])
second = pd.Series([2, 4, 6], index = ['a','b','c'])
print(first[0])
## 2
print(second['c'])
## 6
```

2.2.3 Vectorization in R

An operation is **vectorized** if it applies to all of the elements of a `vector` at once. An operator that is not vectorized can only be applied to individual elements. In that case, the programmer would need to write more code to instruct the function to be applied to all of the elements of a vector. You should prefer writing vectorized code because it is easier to read, and quite often it runs much faster.

Arithmetic (e.g. `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`, etc.) and logical (e.g. `!`, `|`, `&`, `>`, `>=`, `<`, `<=`, `==`, etc.) operators are commonly applied on single vectors or between two vectors. Numeric vectors are converted to logical vectors if they need to be. Many functions work on vectors **element-wise** as well (except functions like `sum` or `length`, etc.). Operator precedence is important to understand if you seek to minimize your use of parentheses. Here are some examples.

```
(1:3) * (1:3)
## [1] 1 4 9
(1:3) == rev(1:3)
## [1] FALSE  TRUE FALSE
sin( (2*pi/3)*(1:5))
## [1]  8.660e-01 -8.660e-01 -2.449e-16  8.660e-01
## [5] -8.660e-01
```

In the last example, there is **recycling** happening. `(2*pi/3)` is taking three length-one vectors and producing another length-one vector. That gets multiplied by length five vector `1:5`. The single element in the length one vector gets recycled so that its value is multiplied by every element of `1:5`. This makes sense most of the time, but sometimes it can be tricky. Notice that this does not produce an error–just a warning.

```
(1:3) * (1:4)
## Warning in (1:3) * (1:4): longer object length is not a
## multiple of shorter object length
## [1] 1 4 9 4
```

2.2.4 Vectorization in Python

The Python's Numpy library makes extensive use of vectorization as well. Vectorization in Numpy is accomplished with **universal functions**, or `ufunc` for short. Some `ufuncs` can be invoked using the same syntax as in R (e.g. `+`). You can also refer to function by name (e.g. `np.sum`). Mixing and matching is allowed, too.

`ufuncs` are called *unary* if they take in one array, and *binary* if they take in two. I reproduce a portion of the very nice table found in (VanderPlas, 2016)

Operator	Equivalent `ufunc`
`+`	`np.add`
`-`	`np.subtract`
`-`	`np.negative`
`*`	`np.multiply`

Operator	Equivalent ufunc
/	np.divide
//	np.floor_divide
**	np.power
%	np.mod

For an exhaustive list of Numpy's universal functions, click here. Here are some examples.

```
np.arange(1,4)*np.arange(1,4)
## array([1, 4, 9])
np.zeros(5) > np.arange(-3,2)
## array([ True,  True,  True, False, False])
np.exp( -.5 * np.linspace(-3, 3, 10)**2) / np.sqrt( 2 * np.pi)
## array([0.00443185, 0.02622189, 0.09947714, 0.24197072, 0.37738323,
##        0.37738323, 0.24197072, 0.09947714, 0.02622189, 0.00443185])
```

Instead of calling it "recycling", Numpy calls it **broadcasting**. It's the same idea as in R, but in general, Python is stricter and disallows more scenarios.

Then there are the `Series` objects from `Pandas`. ufuncs still work on `Series` objects, and they respect common index values.

```
s1 = pd.Series(np.repeat(100,3))
s2 = pd.Series(np.repeat(10,3))
s1 + s2
## 0    110
## 1    110
## 2    110
## dtype: int64
```

If you feel more comfortable, and you want to coerce these `Series` objects to Numpy arrays, you can do that. For example, the following works.

```
s = pd.Series(np.linspace(-1,1,5))
np.exp(s.to_numpy())
## array([0.36787944, 0.60653066, 1.        , 1.64872127, 2.71828183])
```

In addition, there are many attributes and methods they possess.

```
ints = pd.Series(np.arange(10))
ints.abs()
## 0    0
## 1    1
## 2    2
## 3    3
## 4    4
## 5    5
## 6    6
## 7    7
## 8    8
## 9    9
## dtype: int64
ints.mean()
## 4.5
ints.floordiv(2)
## 0    0
## 1    0
## 2    1
## 3    1
## 4    2
## 5    2
## 6    3
## 7    3
## 8    4
## 9    4
## dtype: int64
```

`Series` objects that have text data are a little bit different. For one, you have to access the `.str` attribute of the `Series` before calling any vectorized methods. Here is an example.

```
s = pd.Series(['a','b','c','33'])
s.dtype
```

```
## dtype('O')
```

```
s.str.isdigit()
```

```
## 0    False
## 1    False
## 2    False
## 3     True
## dtype: bool
```

```
s.str.replace('a', 'z')
```

```
## 0     z
## 1     b
## 2     c
## 3    33
## dtype: object
```

String operations can be a big game changer.

2.2.5 Indexing vectors in R

It is very common to want to extract or modify a subset of elements in a vector. There are a few ways to do this. All of them involve the square bracket operator (e.g. `[]`). Feel free to retrieve the documentation by typing `?'['`.

```
allElements <- 1:6
allElements[seq(2,6,2)] # extract evens
## [1] 2 4 6
allElements[-seq(2,6,2)] <- 99 # replace all odds with 99
allElements[allElements > 2] # get nums bigger than 20
## [1] 99 99  4 99  6
```

To access the first element, we use the index `1`. To access the second, we use `2`, and so on. Also, the `-` sign tells R to remove elements. Both of these functionalities are *very different* from Python, as we will see shortly.

We can use names to access elements elements, too, but only if the elements are named.

```
sillyVec <- c("favorite"=1, "least favorite" = 2)
sillyVec['favorite']
## favorite
##        1
```

2.2.6 Indexing Numpy arrays

Indexing Numpy arrays is very similar to indexing vectors in R. You use the square brackets, and you can do it with logical arrays or index arrays. There are some important differences, though.

For one, indexing is 0-based in Python. The `0`th element is the first element of an array. Another key difference is that the `-` isn't used to remove elements like it is in R, but rather to count backwards. Third, using one or two `:` inside square brackets is more flexible in Python. This is syntactic sugar for using the `slice()` function, which is similar to R's `seq()` function.

```
one_through_ten = np.arange(1, 11)
one_through_ten[np.array([2,3])]
## array([3, 4])
one_through_ten[1:10:2] # evens
## array([ 2,  4,  6,  8, 10])
one_through_ten[::-1] # reversed
## array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
one_through_ten[-2] = 99 # second to last
one_through_ten
## array([ 1,  2,  3,  4,  5,  6,  7,  8, 99, 10])
one_through_ten[one_through_ten > 3] # bigger than three
## array([ 4,  5,  6,  7,  8, 99, 10])
```

2.2.7 Some Gotchas

2.2.7.1 Shallow versus Deep Copies

In R, assignment usually produces a **deep copy.** In the code below, we create `b` from `a`. If we modify `b`, these changes don't affect `a`. This takes up more memory, but our program is easier to follow as we don't have to keep track of connections between objects.

```
# in R
a <- c(1,2,3)
b <- a
b[1] <- 999
a # still the same!
## [1] 1 2 3
```

With Numpy arrays in Python, each copy is typically a **shallow copy.** In the code below, `b` is a **reference** for `a`. They both point to the same data in memory. We make a change to `b`, and it *does* affect `a`. This can make the program more confusing, although it can improve computational efficiency. More details on how Python treats the assignment operator can be found in 2.5.7.2.

```
# in python
a = np.array([1,2,3])
b = a
b[0] = 999
a # two names for the same object in memory
## array([999,   2,   3])
```

If you want a deep copy in Python, use `np.copy`.

```
# in python
a = np.array([1,2,3])
b = np.copy(a)
b[0] = 999
a
## array([1, 2, 3])
```

2.2.8 How do R and Python handle missing values?

R has `NULL`, `NaN`, and `NA`. Python has `None`, `np.nan`. If your eyes are glazing over already and you're thinking "they all look like the same"–they are not.

R's `NULL` and Python's `None` are similar. Both represent "nothingness." This is *not* the same as `0`, or an empty string, or `FALSE`/`False`. This is commonly used to detect if a user fails to pass in an argument to a function, or if a function fails to "return" (more information on functions can be found in section 2.5) anything meaningful.

In R, for example, if a function fails to return anything, then it actually returns a `NULL`.

```
NULL==FALSE
## logical(0)
NULL==NULL
## logical(0)
doNothingFunc <- function(a){}
thing <- doNothingFunc()
is.null(thing)
## [1] TRUE
typeof(NULL)
## [1] "NULL"
```

In Python, we have the following.

```
None == False
## False
None == None
## True
def do_nothing_func():
  pass
thing = do_nothing_func()
if thing is None:
  print("thing is None!")
## thing is None!
type(None)
## <class 'NoneType'>
```

`NaN` stands for “not a number.” It is an object of type `double` in R, and type `float` in Python. It can come in handy when you have 0/0 or $\infty / -\infty$.

```
# in R
0/0
## [1] NaN
Inf/Inf
## [1] NaN
is.na(0/0)
## [1] TRUE
```

```
# in Python
0/0
## Error in py_call_impl(callable, dots$args, dots$keywords): ZeroDivisionError: di
import numpy as np
np.inf/np.inf
## nan
np.isnan(np.nan)
## True
```

“NA” is short for “not available.” Missing data is a fact of life in data science. Observations are often missing in data sets, or introduced after joining/merging data sets together (more on this in section 3.4.3). There are many techniques designed to estimate quantities in the presence of missing data. When you code them up, you’ll need to make sure you deal with `NA`s properly.

```
# in R
babyData <- c(0,-1,9,NA,21)
NA == TRUE
## [1] NA
is.na(babyData)
## [1] FALSE FALSE FALSE  TRUE FALSE
typeof(NA)
## [1] "logical"
```

Unfortunately Pythons support of an `NA`-like object is more limited. There is no `NA` object in base Python. And often `NaN`s will appear in place of an

NA. There are a few useful tools, though. The Numpy library offers "masked arrays", for one.

Also, as of version 1.0.0, the Pandas library has an experimental pd.NA object. However, they warn that "the behaviour of pd.NA can still change without warning."

```
import numpy as np
import numpy.ma as ma
baby_data = ma.array([0,-1,9,-9999, 21]) # -9999 "stands for" missing
baby_data[3] = ma.masked
np.average(baby_data)
## 7.25
```

Be careful of using extreme values to stand in for what should be an NA. Failing to mark the above missing value correctly would lead to extremely wrong calculations!

2.3 Numpy's ndarrays versus R's matrices and arrays

Sometimes you want a collection of elements that are all the same type, but you want to store them in a two- or three-dimensional structure. For instance, say you need to use matrix multiplication for some linear regression software you're writing, or that you needed to use tensors for a computer vision project you're working on.

2.3.1 In Python

In Python, you could still use arrays for these kinds of tasks. You will be pleased to learn that the Numpy arrays we discussed earlier are a special case of Numpy's N-dimensional arrays. Each array will come with an enormous amount of methods and attributes (more on object-oriented program in chapter 4) attached to it. A few are demonstrated below.

```
import numpy as np
a = np.array([[1,2],[3,4]], np.float)
a
## array([[1., 2.],
##        [3., 4.]])
a.shape
## (2, 2)
a.ndim
## 2
a.dtype
## dtype('float64')
a.max()
## 4.0
a.resize((1,4)) # modification is **in place**
a
## array([[1., 2., 3., 4.]])
```

2.3.2 In R

TODO (Matloff, 2011)

2.4 R's lists versus Python's lists and dictionaries

When you need to store elements in a container, but you can't guarantee that these elements all have the same type, or you can't guarantee that they all have the same size, then you need a `list` in R. In Python, you might need a `list` or `dict` (short for dictionary).

2.4.1 Lists In R

`lists` are one of the most flexible data types in R. You can access individual elements in many different ways, each element can be of different size, and each element can be of a different type.

```
myList <- list(c(1,2,3), "May 5th, 2021", c(TRUE, TRUE, FALSE))
myList[1] # length-1 list; first element is length 3 vector
## [[1]]
## [1] 1 2 3
myList[[1]] # length-3 vector
## [1] 1 2 3
```

If you want to extract an element, you need to decide between using single square brackets or double square brackets. The former returns a `list`, while the second returns the type of the individual element.

You can also name the elements of a list. This can lead to more readable code. To see why, examine the example below. The `lm()` function estimates a linear regression model. It returns a `list` with plenty of components.

```
dataSet <- read.csv("data/cars.csv")
results <- lm(log(Horsepower) ~ Type, data = dataSet)
length(results)
names(results)
results$contrasts
results['rank']
results[['terms']]
```

2.4.2 Lists In Python

Python `lists` are very flexible, too. There are fewer choices for accessing elements of lists in Python–you'll most likely end up using the square bracket operator. Elements can be different sizes and types, just like they were with R's `lists`.

Unlike in R, however, you cannot name elements of lists. If you want a container that allows you to access elements by name, look into Python dictionaries (see section 2.4.3) or Pandas `Series` objects (see section 2.2.2).

From the example below, you can see that we've been introduced to lists already. We have been constructing Numpy arrays from them.

```
another_list = [np.array([1,2,3]), "May 5th, 2021", True, [42,42]]
another_list[2]
## True
```

2.4.3 Dictionaries In Python

Dictionaries in Python provide a container of key-value pairs. The keys are *unique*, and they must be *immutable*. `string`s are the most common key type, but `int`s can be used as well.

Here is an example of creating a `dict` that stores the current price of a few popular cryptocurrencies. Accessing an individual element's value using its key is dont with the square bracket operator (i.e. `[]`), and deleting elements is done with the `del` keyword.

```
current_crypto_prices = {'BTC': 38657.14, 'ETH': 2386.54, 'DOGE': .308122}
current_crypto_prices['DOGE'] # get the current price of Dogecoin
## 0.308122
del current_crypto_prices['BTC'] # remove the current price of Bitcoin
current_crypto_prices.keys()
## dict_keys(['ETH', 'DOGE'])
current_crypto_prices.values()
## dict_values([2386.54, 0.308122])
```

You can also create `dict`s using **dictionary comprehensions**

```
incr_cryptos = {key:val*1.1 for (key,val) in current_crypto_prices.items()}
incr_cryptos
## {'ETH': 2625.194, 'DOGE': 0.3389342}
```

2.5 User-defined functions in R and Python

Why are functions important in statistical programming? In both R and Python (and every other programming language), functions are used to perform calculations. This is a pretty obvious statement.

This text has already covered how to *use* functions that come to us pre-made. At least we have discussed how to use them in a one-off way–just write the name of the function, write some parentheses after that name, and then plug in any requisite arguments by writing them in a comma-separated way between those two parentheses. This is how it works in both R and Python. In this section we take a look at how to define our own functions. This will help us understand pre-made functions. It will also be useful if we need some functionality that isn't already written for us.

Writing our own functions is also useful for "packaging up" computations. The utility of this will become very apparent in chapter 5. Consider the task of estimating a regression model. Would you want to write that program using only arithmetic operators? Would it be simpler if you could use matrix multiplications? Would you want your function to work on different types of inputs? Would you want it to estimate several regression models and choose the "best" one?

Thankfully, R functions are very similar to Python functions. In both languages, functions are **first-class objects**. This means that, no matter which of these two languages you're using, functions

- can be passed as arguments to other functions,
- they can be returned as values from other functions, and
- they can be assigned to variables and stored in containers (Abelson and Sussman, 1996)

2.5.1 Defining R Functions

To create a function in R, we need another function called `function`. We give the output of `function` a name in the same way we give names to any other variable in R, by using the assignment operator `<-`. Here's an example of a toy function called `addOne`. Here `myInput` is a placeholder that refers to whatever the user of the function ends up plugging in.

```r
addOne <- function(myInput){  # define the function
  myOutput <- myInput + 1
  return(myOutput)
}
```

```
addOne(41) # call/invoke/use the function
## [1] 42
```

Below the definition, the function is called with an input of `41`. When this happens, the following sequence of events occurs

- The value `41` is assigned to `myInput`
- `myOutput` is given the value `42`
- `42` is returned from the function
- the temporary variables `myInput` and `myOutput` are destroyed.

We get the desired answer, and all the unnecessary intermediate variables are cleaned up and thrown away.

2.5.2 Defining Python Functions

To create a function in Python, we use the `def` statement (instead of the `function` function in R). The desired name of the function comes next. After that, the formal parameters come, comma-separated inside parentheses, just like in R.

Defining a function in Python is a little more concise. There is no assignment operator like there is in R, there are no curly braces, and `return` isn't a function like it is in R, so there is no need to use parentheses after it. There is one addition, though–we need a colon (`:`).

Here is an example of a toy function called `add_one`.

```
def add_one(my_input):  # define the function
  my_output = my_input + 1
  return my_output
add_one(41) # call/invoke/use the function
## 42
```

Below the definition, the function is called with an input of `41`. When this happens, the following sequence of events occurs

- The value `41` is assigned to `my_input`
- `my_output` is given the value `42`
- `42` is returned from the function
- the temporary variables `my_input` and `my_output` are destroyed.

We get the desired answer, and all the unnecessary intermediate variables are cleaned up and thrown away.

2.5.3 More details on R's user-defined functions

Technically, in R, functions are defined as three things bundled together:

1. a **formal argument list** (also known as *formals*),
2. a **body**, and
3. a **parent environment**.

The *formal argument list* is exactly what it sounds like. It is the list of arguments a function takes. You can access a function's formal argument list using the `formals()` function. Note that it is not the *actual* arguments a user will plug in–that isn't knowable at the time the function is created in the first place.

Here is another function that takes a *default* argument called `whichNumber`. If the user of the function doesn't specify how much she wants to add to `myInput`, `addNumber` will use `1` as the default. This default value shows up in the output of `formals(addNumber)`.

```
addNumber <- function(myInput, whichNumber = 1){
  myOutput <- myInput + whichNumber
  return(myOutput)
}
addNumber(3) # no second argument being provided by the user here
## [1] 4
formals(addNumber)
## $myInput
##
##
```

```
## $whichNumber
## [1] 1
```

The function's *body* is also exactly what it sounds like. It is the work that a function performs. You can access a functions body using the the `body()` function.

```
addNumber <- function(myInput, whichNumber = 1){
  myOutput <- myInput + whichNumber
  return(myOutput)
}
body(addNumber)
## {
##     myOutput <- myInput + whichNumber
##     return(myOutput)
## }
```

Every function you create also has a *parent environment*[2]. You can get/set this using the `environment()` function. Environments help a function know which variables it is allowed to use and how to use them. The parent environment of a function is where the function was *created*, and it contains variables outside of the body that the function can also use. The rules of which variables a function can use are called *scoping*. When you create functions in R, you are primarily using **lexical scoping**. To understand functions well in R, these examples are important to understand, so I provide more detail in 2.5.5.

There is a lot more information about environments that isn't provided in this text. For instance, a user-defined function also has binding, execution, and calling environments associated with it, and environments are used in creating package namespaces, which are important when two packages each have a function with the same name.

[2]Primitive functions are functions that contain no R code and are internally implemented in C. These are the only type of function in R that don't have a parent environment.

2.5.4 More details on Python's user-defined functions

Python functions have the same things R functions have: a **formal parameter list**, a body, and there are namespaces created that help organize which variables the function can access, as well as which pieces of code can call this new function. These three concepts are analogous to those in R. The names are just a bit different sometimes, and it isn't organized in the same way.

Below is a table, taken straight from the documentation, of all each user-defined function's *special attributes*.

Attribute	Meaning
`__doc__`	The function's documentation string, or `None` if unavailable; not inherited by subclasses.
`__name__`	The function's name.
`__qualname__`	The function's qualified name.
`__module__`	The name of the module the function was defined in, or None if unavailable.
`__defaults__`	A tuple containing default argument values for those arguments that have defaults, or None if no arguments have a default value.
`__code__`	The code object representing the compiled function body.
`__globals__`	A reference to the dictionary that holds the function's global variables — the global namespace of the module in which the function was defined.
`__dict__`	The namespace supporting arbitrary function attributes.
`__closure__`	`None` or a tuple of cells that contain bindings for the function's free variables. See below for information on the `cell_contents` attribute.
`__annotations__`	A dict containing annotations of parameters. The keys of the dict are the parameter names, and 'return' for the return annotation, if provided.
`__kwdefaults__`	A dict containing defaults for keyword-only parameters.

So take the *formal parameter list* of a user-defined function, which is, again, the list of inputs a function takes. Just like in R, this is not the *actual* arguments a user will plug in–that isn't knowable at the time the function is created.[3] Below we have another function that takes a **default argument** called `which_number`. If the user of the function doesn't specify how much she wants to add to `my_input`, `add_number` will use `1` as the default. This default value can be obtained with `add_number.__defaults__`.

```python
def add_number(my_input, which_number = 1):
  my_output = my_input + which_number
  return my_output
add_number(3) # no second argument being provided by the user here
## 4
add_number.__code__.co_varnames # this also contains my_output
## ('my_input', 'which_number', 'my_output')
add_number.__defaults__
## (1,)
```

The code attribute has much more to offer. To see a list of names of all its contents, you can use `dir(add_number.__code__)`.

Don't worry if the notation `add_number.__code__` looks strange. The dot (`.`) operator will become more clear in the future chapter on *object-oriented programming*. For now, just think of `__code__` as being an object *belonging to* `add_number`. Objects that belong to other objects are called **attributes** in Python. The dot operator helps us access attributes *inside* other objects.

2.5.5 Function Scope in R

R uses **lexical scoping**.

R functions can use variables that are defined in the function body, and variables that were defined in the environment that the function itself was defined in. R functions **cannot** necessarily find variables in an environment

[3]You might have noticed that Python uses two different words to prevent confusion–unlike R, Python uses the word "parameter" (instead of "argument") to refer to the inputs a function takes, and "arguments" to the specific values a user plugs in.

where the function was *called* in. Code outside the body of a function cannot access variables inside the body of a function.

```
a <- 3
sillyFunction <- function(){
  return(a + 20)
}
environment(sillyFunction) # the env. it was defined in contains a
## <environment: R_GlobalEnv>
sillyFunction()
## [1] 23
```

From the point of view of the function, when it attempts to access a variable, it first looks in its own body. In the example below, there are two variables named `a`, but they exist in different environments. Inside the function, the innermost one gets used. Outside the function, the global variable gets used.

```
a <- 3
sillyFunction <- function(){
  a <- 20
  return(a + 20)
}
sillyFunction()
## [1] 40
print(a)
## [1] 3
```

The same concept applies if you create functions within functions. The inner function looks “inside-out” for variables. Below we call `outerFunc()`, which calls `innerFunc()`. `innerFunc()` can refer to the variable `b`, because it lies in the same environment in which `innerFunc()` was created. Interestingly, `innerFunc()` can also refer to the variable `a`, because that variable was captured by `outerFunc`, which provides access to `innerFunc`.

```
a <- "outside both"
outerFunc <- function(){
  b <- "inside one"
  innerFunc <- function(){
    print(a)
    print(b)
  }
  return(innerFunc())
}
outerFunc()
## [1] "outside both"
## [1] "inside one"
```

If we ask `outerFunc` to return the function `innerFunc` (functions are objects!), then we might be surprised to see that `innerFunc()` can still successfully refer to `b`, even though it doesn't exist inside the *calling environment*. But don't be surprised! What matters is what was available when the function was *created*. In this example, `outerFuncV2` is sometimes called a *function factory*. More information about this is provided in 5.

```
outerFuncV2 <- function(){
  b <- "inside one"
  innerFunc <- function(){
    print(b)
  }
  return(innerFunc) # note the missing inner parentheses!
}
myFunc <- outerFuncV2() # get a new function
ls(environment(myFunc)) # list all data attached to this function
## [1] "b"         "innerFunc"
myFunc()
## [1] "inside one"
```

Sometimes, in R, functions are called **closures** to emphasize that they are capturing variables from the parent environment in which they were created, to emphasize the data that they are bundled with.

2.5.6 Function Scope in Python

Python uses **lexical scoping** just like R! There's a famous acronym for the concept in Python: **LEGB**.

- L: Local,
- E: Enclosing,
- G: Global, and
- B: Built-in.

A Python function will search for a variable in these namespaces in this order.[4].

"*Local*" refers to variables that are defined inside of the function's block. The function below uses the local `a` over the global one.

```
a = 3
def silly_function():
  a = 22 # local a
  print("local variables are ", locals())
  return a + 20
silly_function()
## local variables are  {'a': 22}
## 42
silly_function.__code__.co_nlocals # number of local variables
## 1
silly_function.__code__.co_varnames # names of local variables
## ('a',)
```

"*Enclosing*" refers to variables that were defined in the enclosing namespace, but not the global namespace. These variables are sometimes called **free variables.** In the example below, there is no local `a` variable for `inner_func`. But there is a global one and one in the enclosing namespace. It chooses the one in the enclosing namespace.

[4]Functions aren't the only thing that get their own namespace. For instance, classes do as well. More information on classes is provided in Chapter 4

```
a = "outside both"
def outer_func():
  a = "inside one"
  def inner_func():
    print(a)
  return inner_func
my_new_func = outer_func()
my_new_func()
## inside one
my_new_func.__code__.co_freevars
## ('a',)
```

"*Global*" scope contains variables defined in the module-level namespace. If the below example code was the entirety of your script, then `a` would be a global variable.

```
a = "outside both"
def outer_func():
  b = "inside one"
  def inner_func():
    print(a)
  inner_func()
outer_func()
## outside both
```

Just like in R, Python functions **cannot** necessarily find variables in an environment where the function was *called* in. For example, here is some code that mimics the above R example. Both `a` and `b` are accessible from within `inner_func`. That is due to LEGB.

```
a = "outside both"
def outer_func():
  b = "inside one"
  def inner_func():
    print(a)
    print(b)
  return inner_func()
outer_func()
```

```
## outside both
## inside one
```

However, if we start using `outer_func` inside another function, *calling* it in another function, when it was *defined* somewhere else, well then it doesn't have access to some variables. You might be surprised at how the following code functions. Does this print the right string: `"this is the a I want to use now!"` No!

```
def third_func():
  a = "this is the a I want to use now!"
  outer_func()
third_func()
```

```
## outside both
## inside one
```

Again, these examples get at *functional programming*, which is discussed more in depth in chapter 5. There it will describe strategies to make your code easier to maintain (e.g. keep your functions "pure"!)

2.5.7 Modifying a Function's Arguments

Can/should we modify a function's argument? The flexibility to do this sounds empowering; however, not doing it is recommended because it makes programs easier to reason about.

2.5.7.1 Passing By Value In R

In R, it is *difficult* for a function to modify the variable that a user plugs in to a function as its argument.[5] Consider the following code.

[5]There are some exceptions to this, but it's generally true.

```
a <- 1
f <- function(arg){
  arg <- 2
  return(arg)
}
print(a)
## [1] 1
print(f(a))
## [1] 2
print(a)
## [1] 1
```

The function `f` has an argument called `arg`. When `f(a)` is performed, changes are made to a *copy* of `a`. When a function constructs a copy of all input variables inside its body, this is called **pass-by-value** semantics. This copy is a temporary intermediate value that only serves as a starting point for the function to produce a return value of `2`.

`arg` could have been called `a`, and the same behavior will take place. However, giving these two things different names is helpful to remind you and others that R copies its arguments.

It is still possible to modify `a`, but I don't recommend doing this either. I will discuss this more in subsection 2.5.7.

2.5.7.2 Passing By Assignment In Python

The story is more complicated in Python. Python functions have **pass-by-assignment** semantics. This is something that is very unique to Python. What this means is that your ability to modify the arguments of a function depends on

- what the type of the argument is, and
- what you're trying to do to it.

We will go throw some examples first, and then explain why this works the way it does. Here is some code that is analogous to the example above.

```python
a = 1
def f(arg):
  arg = 2
  return arg

print(type(a))
## <class 'int'>
print(a)
## 1
print(f(a))
## 2
print(a)
## 1
```

In this case, `a` is not modified. That is because `a` is an `int`. `int`s are **immutable** in Python, which means that their value cannot be changed after they are created, either inside or outside of the function's scope. However, consider the case when `a` is a `list`, which is a **mutable** type. A mutable type is one that can have its value changed after its created.

```python
a = [999]
def f(arg):
  arg[0] = 2
  return arg

print(type(a))
## <class 'list'>
print(a)
## [999]
print(f(a))
## [2]
print(a)
## [2]
```

In this case `a` *is* modified. Changing the value of the argument *inside* the function effects changes to that variable outside of the function.

Ready to be confused? What happens if we take in a list, but try to do something else with it.

```
a = [999]
def f(arg):
  arg = [2]
  return arg

print(a)
## [999]
print(f(a))
## [2]
print(a)
## [999]
```

That time `a` did not permanently change in the global scope. Why does this happen? I thought `lists` were mutable!

The reason behind all of this doesn't even have anything to do with functions, per se. Rather, it has to do with how Python manages, objects, values, and types. It also has to do with what happens during assignment.

Let's revisit the above code, but bring everything out of a function. Python is pass-by-assignment, so all we have to do is understand how assignment works. Starting with the immutable `int` example, we have the following.

```
# old code:
# a = 1
# def f(arg):
#   arg = 2
#   return arg
a = 1    # still done in global scope
arg = a  # arg is a name that is bound to the object a refers to
arg = 2  # arg is a name that is bound to the object 2
print(arg is a)
## False
print(id(a), id(arg))
## 4520806896 4520806928
```

```
print(a)
## 1
```

The `id()` function returns the **identity** of an object, which is kind of like its memory address. Identities of objects are unique and constant. If two variables, `a` and `b` say, have the same identity, `a is b` will evaluate to `True`. Otherwise, it will evaluate to `False`.

In the first line, the *name* `a` is bound to the *object* `1`. In the second line, the name `arg` is bound to the *object* that is referred to by the *name* `a`. After the second line finishes, `arg` and `a` are two names for the same object (a fact that you can confirm by inserting `arg is a` immediately after this line).

In the third line, `arg` is bound to `2`. The variable `arg` can be changed, but only by re-binding it with a separate object. Re-binding `arg` does not change the value referred to by `a` because `a` still refers to `1`, an object separate from `2`. There is no reason to re-bind `a` because it wasn't mentioned at all in the third line.

If we go back to the first function example, it's basically the same idea. The only difference, however, is that `arg` is in its own scope. Let's look at a simplified version of our second code chunk that uses a mutable list.

```
a = [999]
# old code:
# def f(arg):
#   arg[0] = 2
#   return arg
arg = a
arg[0] = 2
print(arg)
## [2]
print(a)
## [2]
print(arg is a)
## True
```

In this example, when we run `arg = a`, the name `arg` is bound to the same object that is bound to `a`. This much is the same. The only difference here, though, is that because lists are mutable, changing the first element of `arg` is done "in place", and all variables can access the mutated object.

Why did the third example produce unexpected results?

```
a = [999]
# old code
# def f(arg):
#   arg = [2]
#   return arg
arg = a
arg = [2]
print(arg is a)
## False
print(a)
## [999]
print(arg)
## [2]
```

The difference is in the line `arg = [2]`. This rebinds the name `arg` to a different variable. `lists` are still mutable, but this has nothing to do with re-binding–re-binding a name works no matter what type of object you're binding it to. In this case we are re-binding `arg` to a completely different list.

2.5.8 Accessing and Modifying Non-Local Variables

In the last subsection, we were talking about variables that were passed in as arguments to a function. Here we are talking about variables that are not, but are still referred to inside a function's body.

In general, even though it is possible to access and modify non-local variables in both languages, it is not a good idea.

2.5.8.1 Accessing and Modifying Non-Local Variables in R

As Hadley Wickham writes in his book, "[l]exical scoping determines where, but not when to look for values." R has **dynamic lookup**, meaning code inside a function will only try to access a referred-to variable when the function is *running*, not when it is defined.

Consider the R code below.

```r
# R
missileLaunchCodesSet <- TRUE
everythingIsSafe <- function(){
  return(!missileLaunchCodesSet)
}
missileLaunchCodesSet <- FALSE
# everythingIsSafe() # what happens if we call it?
```

`everythingIsSafe` is created in the global environment, and the global environment contains a Boolean variable called `missileLaunchCodesAreSet`.

Now imagine sharing some code with a collaborator. Imagine, further, that your collaborator is the subject-matter expert, and knows little about R programming. Suppose that he changes a global variable in the script. Shouldn't this induce a relatively trivial change to the overall program?

Let's explore this hypothetical further. Consider what could happen if any of the following (very typical) conditions are true:

- you or your collaborators aren't sure what `everythingIsSafe` will return because you don't understand dynamic lookup, or
- it's difficult to visually keep track of all assignments to `missileLaunchCodesAreSet` (e.g. your script is quite long or it changes often), or
- you are not running code sequentially (e.g. you are testing chunks at a time instead of clearing out your memory and `source()`ing from scratch, over and over again).

In each of these situations, understanding of the program would be compromised. However, if you follow the above principle of never referring to non-local variables in function code, all members of the group could do their own work separately, minimizing the dependence on one another.

Another reason violating this could be troublesome is if you define a function that refers to a nonexistent variable. *Defining* the function will never throw an error because R will assume that variable is defined in the global environment. *Calling* the function might throw an error, unless you accidentally defined the variable, or if you forgot to delete a variable whose name you no longer want to use.

```
# R
myFunc <- function(){
  return(varigbleNameWithTypo)
}
```

Running the above code to define `myFunc` will not throw an error, even if you think it should!

2.5.8.2 Accessing and Modifying Non-Local Variables in Python

It is the same exact situation in Python. Consider `everything_is_safe`, a function that is analogous to `everythingIsSafe`.

```
# python
missile_launch_codes_set = True
def everything_is_safe():
  return not missile_launch_codes_set

missile_launch_codes_set = False
everything_is_safe()
## True
```

We can also define `my_func`, which is analogous to `myFunc`. Defining this function doesn't throw an error either!

```
# python
def my_func():
  return varigble_name_with_typo
```

So stay away from referring to variables outside the body of your function!

2.5.8.3 Modifying Non-Local Variables In R

Now what if we want to be extra bad, and in addition to *accessing* global variables, we *modify* them, too.

```
a <- 1
f <- function(arg){
  arg <- 2
  a <<- arg
  # return(arg) # no return value
}
print(a)
## [1] 1
print(f(a))
## [1] 2
print(a)
## [1] 2
```

In the program above, `arg` creates a copy of `a`. It assigns `2` to that copy. Then it takes that `2` and writes it to the global variable in the parent environment. Notice that the function can take in different inputs, but the global assignment is hard-coded.

2.5.8.4 Modifying Non-Local Variables In Python

Finally, there is something in Python that is like R's super assignment operator (`<<-`). It is the `global` keyword. This will let you *modify* global variables.

Referring to global variables *without* modifying them was always allowed, even without using the `global` keyword. This keyword should be used sparingly, and when it is used, it identifies that a function causes **side effects**, which are changes in some variable defined outside of the function's scope.

```
a = 1
def increment_a():
  global a
  a += 1
increment_a()
increment_a()
increment_a()
print(a)
## 4
```

Here's a last example that will be important for us in particular. Notice that Numpy `array`s are mutable.

```
import numpy as np
my_array = np.array([1,2,3])
def make_calc(arr):
  arr[0] = np.average(my_array)
  return 2*arr
result = make_calc(my_array)
print(result)
## [4 4 6]
print(my_array) # watch out: side effect
## [2 2 3]
```

2.6 Categorical Data

2.6.1 Categorical Data in R

Categorical data is typically stored in a `factor` variable in R. For example, say we asked three people what their favorite season was. The data might look something like this.

```
responses <- factor(c("autumn", "summer", "summer"),
                    levels = c("autumn", "summer", "spring", "winter"))
levels(responses)
## [1] "autumn" "summer" "spring" "winter"
contrasts(responses)
##        summer spring winter
## autumn      0      0      0
## summer      1      0      0
## spring      0      1      0
## winter      0      0      1
is.factor(responses)
## [1] TRUE
is.ordered(responses)
## [1] FALSE
```

`factor`s have a `levels` attribute, which is comprised of all the possible values that each response could be. They also have a `contrasts` attribute, which will be important once you start using `factor`s as inputs to functions such as `lm`. In the case of using `factor`s as inputs to `lm()`, the `factor` would tell `lm()` *how* to create the dummy predictors in a linear regression model. It's perfectly fine if you're rusty on regression–the reason I mention this is that in Python, dummy variable construction is done more explicitly/manually.

In the above example, there wasn't at least one person who prefers each season (that's a confusing sentence). Here, if we did not specify a `levels` argument, there would only be two levels. This is a common source of bugs! Another source of bugs: what if some people say "autumn" and others say "fall"?

`factor`s can be ordered or unordered. Ordered `factor`s are for ordinal data. As another example, say we asked ten people how much they liked programming, and they could only respond "love it", "hate it", or "it's okay". The data might look something like this.

```
responses <- factor(c("love it", "it's okay", "love it",
                      "love it", "it's okay", "love it",
                      "love it", "love it", "it's okay",
                      "it's okay"),
                    levels = c("hate it", "it's okay", "love it"),
                    ordered = TRUE)
levels(responses)
## [1] "hate it"   "it's okay" "love it"
contrasts(responses)
##              .L      .Q
## [1,] -7.071e-01  0.4082
## [2,] -7.850e-17 -0.8165
## [3,]  7.071e-01  0.4082
is.factor(responses)
## [1] TRUE
is.ordered(responses)
## [1] TRUE
```

Whether a `factor` is ordered or not can affect its `contrasts` and the behavior of functions it is fed into. Intuitively, it should be clear when to impose ordering or not. In the first example, there isn't a clear ordering of the seasons (which one should come first?). In the second example, we are looking at responses to a "how much" question.

Here's a third example. We can take non-categorical data, and `cut` it into something categorical.

```
stockReturns <- rnorm(10) # not categorical here
typeOfDay <- cut(stockReturns, breaks = c(-Inf, 0, Inf))
typeOfDay
##  [1] (-Inf,0] (-Inf,0] (0, Inf] (0, Inf] (-Inf,0]
##  [6] (0, Inf] (-Inf,0] (-Inf,0] (0, Inf] (0, Inf]
## Levels: (-Inf,0] (0, Inf]
levels(typeOfDay)
## [1] "(-Inf,0]" "(0, Inf]"
is.factor(typeOfDay)
## [1] TRUE
is.ordered(typeOfDay)
## [1] FALSE
```

2.6.2 Categorical Data in Python

Categorical data can be handled with the Pandas' library, which takes a lot of inspiration from R. We've talked about `Series` objects before in section 2.2.4, and here we will use them again. All we have to do to make a `Series` object categorical is to change its `dtype`. The `dtype` we provide will control the categories (like `levels` in R), and whether it's ordered or not.

```
import pandas as pd
from pandas.api.types import CategoricalDtype

cat_type = CategoricalDtype(categories=["autumn", "summer", "spring", "winter"
                            ordered=False)
responses = pd.Series(["autumn", "summer", "summer"],
                      dtype = cat_type)
responses
## 0    autumn
## 1    summer
## 2    summer
## dtype: category
## Categories (4, object): ['autumn', 'summer', 'spring', 'winter']
responses.cat.categories
## Index(['autumn', 'summer', 'spring', 'winter'], dtype='object')
responses.cat.ordered
## False
```

Pandas also provides a `pd.cut()` function, which can return either of these types, or even a regular Numpy array.

```
stock_returns = np.random.normal(size=10) # not categorical here
type_of_day = pd.cut(stock_returns, [-np.inf, 0, np.inf], labels = ['bad day',
type_of_day
## ['bad day', 'good day', 'good day', 'good day', 'bad day', 'bad day', 'ba
## Categories (2, object): ['bad day' < 'good day']
type(type_of_day)
## <class 'pandas.core.arrays.categorical.Categorical'>
type_of_day = pd.Series(type_of_day)
type(type_of_day)
## <class 'pandas.core.series.Series'>
type_of_day.cat.categories
## Index(['bad day', 'good day'], dtype='object')
type_of_day.cat.ordered
## True
```

You'll notice that, in this instance, `pd.cut` did not return a `Series` object. It can, but `pd.cut`'s return type will depend on the inputs you feed in. In this case, it returned a `Categorical`, which is not the same thing as a `Series`. In the code above, I had to convert it back before accessing the `cat` attribute.

2.7 Data Frames

As data scientists, most of the time, our data set will be stored as a data frame.

2.7.1 Data Frames in R

Let's consider as an example Fisher's "Iris" data set. We will read this data set in from a comma separated file (more on input/output in chapter 3.1). This file can be downloaded from this link: https://archive.ics.uci.edu/ml/datasets/iris.

```
irisData <- read.csv("data/iris.csv", header = F)
head(irisData)
##    V1  V2  V3  V4          V5
## 1 5.1 3.5 1.4 0.2 Iris-setosa
## 2 4.9 3.0 1.4 0.2 Iris-setosa
## 3 4.7 3.2 1.3 0.2 Iris-setosa
## 4 4.6 3.1 1.5 0.2 Iris-setosa
## 5 5.0 3.6 1.4 0.2 Iris-setosa
## 6 5.4 3.9 1.7 0.4 Iris-setosa
typeof(irisData)
## [1] "list"
class(irisData) # we'll talk more about classes later
## [1] "data.frame"
dim(irisData)
## [1] 150   5
nrow(irisData)
## [1] 150
ncol(irisData)
## [1] 5
```

Do not rely on the default arguments of `read.csv` or `read.table`! After you read in a data frame, always check the first few rows to make sure that

1. The number of columns is correct because the correct column *separator* was used (c.f. `sep=`),
2. column names were parsed correctly, if there were some in the raw text file,
3. the first row of data wasn't used as a column name sequence, if there weren't column names in the text file, and
4. the last few rows aren't reading in empty spaces
5. character columns are read in correctly (c.f. `stringsAsFactors=`), and
6. special characters signifying missing data were correctly identified (c.f. `na.strings=`).

There are some exceptions, but most data sets can be stored as a `data.frame`. This is because usually a data set comes in a two-dimensional shape Looking at one particular row gives you an observation with all its variables. Looking at an particular column gives you one particular variable for each observation.

A `data.frame` is a special case of a `list`. Every element of the list is a column. Columns can be `vectors` or `factors`, and they can all be of a different type. This is one of the biggest differences between data frames and `matrixs`. They are both two-dimensional, but a `matrix` needs elements to be all the same type. Unlike a general `list`, a `data.frame` requires all of its columns to have the same number of elements. In other words, the `data.frame` is not a "*ragged*" list.

Often times you will need to extract pieces of information from a `data.frame`. This can be done in many ways. If the columns have names, you can use the `$` operator to access a single column. Accessing a single column might be followed up by creating a new vector. You can also use the `[` operator to access multiple columns by name.

```
colnames(irisData) <- c("sepal.length", "sepal.width", "petal.length","petal.w
firstCol <- irisData$sepal.length
head(firstCol)
## [1] 5.1 4.9 4.7 4.6 5.0 5.4
firstTwoCols <- irisData[c("sepal.length", "sepal.width")]
```

```
head(firstTwoCols)
##   sepal.length sepal.width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
## 6          5.4         3.9
```

The `[` operator is also useful for selecting rows and columns by index numbers, or by some logical criteria.

```
topLeft <- irisData[1,1] # first row, first col
topLeft
## [1] 5.1
firstThreeRows <- irisData[1:3,] # rows 1-3, all cols
firstThreeRows
##   sepal.length sepal.width petal.length petal.width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
##       species
## 1 Iris-setosa
## 2 Iris-setosa
## 3 Iris-setosa
setosaOnly <- irisData[irisData$species == "Iris-setosa",] # rows where species colu
head(setosaOnly)
##   sepal.length sepal.width petal.length petal.width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
## 4          4.6         3.1          1.5         0.2
## 5          5.0         3.6          1.4         0.2
## 6          5.4         3.9          1.7         0.4
##       species
## 1 Iris-setosa
## 2 Iris-setosa
## 3 Iris-setosa
## 4 Iris-setosa
## 5 Iris-setosa
## 6 Iris-setosa
```

In the code above, `irisData$species == "Iris-setosa"` creates a logical vector (try it!) using the vectorized `==` operator. The `[` operator selects the rows for which the corresponding element of this logical vector is `TRUE`.

Be careful: depending on how you use the square brackets, you can either get a `data.frame` or a `vector`. As an example, try both `class(irisData[,1])` and `class(irisData[,c(1,2)])`.

In R, `data.frame`s have row names. You can get/set this character `vector` with the `rownames()` function. You can access rows by name using the square bracket operator. Personally, I don't typically use this functionality that often.

```
head(rownames(irisData))
## [1] "1" "2" "3" "4" "5" "6"
rownames(irisData) <- as.numeric(rownames(irisData)) + 1000
head(rownames(irisData))
## [1] "1001" "1002" "1003" "1004" "1005" "1006"
irisData["1002",]
##      sepal.length sepal.width petal.length petal.width
## 1002          4.9           3          1.4         0.2
##          species
## 1002 Iris-setosa
```

2.7.2 Data Frames in Python

The Pandas library in Python has data frames that are modeled after R's.

```
import pandas as pd
iris_data = pd.read_csv("data/iris.csv", header = None)
iris_data.head()
##      0    1    2    3            4
## 0  5.1  3.5  1.4  0.2  Iris-setosa
## 1  4.9  3.0  1.4  0.2  Iris-setosa
## 2  4.7  3.2  1.3  0.2  Iris-setosa
## 3  4.6  3.1  1.5  0.2  Iris-setosa
## 4  5.0  3.6  1.4  0.2  Iris-setosa
iris_data.shape
## (150, 5)
len(iris_data) # num rows
## 150
len(iris_data.columns) # num columns
## 5
iris_data.dtypes
## 0    float64
## 1    float64
## 2    float64
## 3    float64
## 4     object
## dtype: object
```

The structure is very similar to that of R's data frame. It's two dimensional, and you can access columns and rows by name or number. Each column is a `Series` object, and each column can have a different `dtype`, which is analogous to R's situation. Again, because the elements need to be the same type along columns only, this is a big difference between 2-d Numpy `arrays` and `DataFrames`.

Just like in R, you can access columns by name. You do that using square brackets. Observe how similar this code is to the corresponding R code above.

```
iris_data.columns = ["sepal.length", "sepal.width", "petal.length",
                     "petal.width", "species"]
first_col = iris_data['sepal.length']
first_col.head()
## 0    5.1
## 1    4.9
## 2    4.7
## 3    4.6
## 4    5.0
## Name: sepal.length, dtype: float64
first_two_cols = iris_data[["sepal.length", "sepal.width"]]
first_two_cols.head()
##    sepal.length  sepal.width
## 0           5.1          3.5
## 1           4.9          3.0
## 2           4.7          3.2
## 3           4.6          3.1
## 4           5.0          3.6
```

Notice that `iris_data['sepal.length']` returns a `Series` and `iris_data[["sepal.length", "sepal.width"]]` returns a Panda's `DataFrame`. This behavior is similar to what happened in R's. For more details, click here.

You can select columns and rows by number with the `.iloc` method. `iloc` is (probably) short for "integer location."

```
# specify rows/cols by number
top_left = iris_data.iloc[0,0]
top_left
## 5.1
first_three_rows = iris_data.iloc[:3,]
first_three_rows
#setosa_only = iris_data[irisData$species == "Iris-setosa",]  # easieriwith
#head(setosaOnly)
##    sepal.length  sepal.width  petal.length  petal.width      species
## 0           5.1          3.5           1.4          0.2  Iris-setosa
## 1           4.9          3.0           1.4          0.2  Iris-setosa
## 2           4.7          3.2           1.3          0.2  Iris-setosa
```

Selecting columns by anything besides integer number can be done with the `.loc()` method. You should generally prefer this method to access columns because accessing things by *name* instead of *number* is more readable. Here are some examples.

```
sepal_w_to_pedal_w = iris_data.loc['sepal.width':'pedal.width']
sepal_w_to_pedal_w
## Empty DataFrame
## Columns: [sepal.length, sepal.width, petal.length, petal.width, species]
## Index: []
setosa_only = iris_data.loc[iris_data['species'] == "Iris-setosa",]
setosa_only.head()
##    sepal.length  sepal.width  petal.length  petal.width      species
## 0           5.1          3.5           1.4          0.2  Iris-setosa
## 1           4.9          3.0           1.4          0.2  Iris-setosa
## 2           4.7          3.2           1.3          0.2  Iris-setosa
## 3           4.6          3.1           1.5          0.2  Iris-setosa
## 4           5.0          3.6           1.4          0.2  Iris-setosa
```

Notice we used a `slice` to access many columns by only referring to the left-most and the right-most. This does not work with the regular square bracket operator. The second example filters out the rows where the `"species"` column elements are equal to `"Iris-setosa"`.

Each `DataFrame` in Pandas comes with an `.index` attribute. This is analogous to a row name in R, but it's much more flexible because the index can take on a variety of types. This can help us highlight the difference between `.loc` and `.iloc`. Recall that `.loc` was label-based selection. Labels don't necessarily have to be strings. Consider the following example

```
iris_data.index
## RangeIndex(start=0, stop=150, step=1)
iris_data = iris_data.set_index(iris_data.index[::-1]) # reverse the index
iris_data.head(2)
##      sepal.length  sepal.width  petal.length  petal.width      species
## 149           5.1          3.5           1.4          0.2  Iris-setosa
## 148           4.9          3.0           1.4          0.2  Iris-setosa
iris_data.tail(2)
##    sepal.length  sepal.width  petal.length  petal.width         species
## 1           6.2          3.4           5.4          2.3  Iris-virginica
## 0           5.9          3.0           5.1          1.8  Iris-virginica
iris_data.loc[0]
## sepal.length               5.9
## sepal.width                    3
## petal.length                 5.1
## petal.width                  1.8
## species           Iris-virginica
## Name: 0, dtype: object
iris_data.iloc[0]
## sepal.length             5.1
## sepal.width              3.5
## petal.length             1.4
## petal.width              0.2
## species           Iris-setosa
## Name: 149, dtype: object
```

`iris_data.loc[0]` selects the `0`th index. The second line reversed the indexes, so this is actually the last row. If you want the first row, use `iris_data.iloc[0]`.

2.7.3 Row Names and Indexes

In Python, Pandas `DataFrame`s have an

```
iris_data.index
## RangeIndex(start=149, stop=-1, step=-1)
```

2.7.4 Getting Versus Setting

2.8 Exercises

All answers to questions related to R should be written in a file named `data_types_exercises.R`. All answers to questions related to Python should be written in a file named `data_types_exercises.py`.

1. Which Python type is appropriate for each piece of data?

a. TODO

b. TODO

2. In R, say you have a vector of prices of some financial asset:

```r
prices <- c(100.10, 95.98, 100.01, 99.87)
```

a. Convert this vector into a vector of *log returns*. Call the variable `log_returns`. If p_t is the price at time t, the log return ending at time t is

$$r_t = \log\left(\frac{p_t}{p_{t-1}}\right) = \log p_t - \log p_{t-1}$$

b. Do the same for *arithmetic returns*. These are regular percent changes if you scale by 100. Call the variable `arith_returns`. The mathematical formula you need is

$$a_t = \left(\frac{p_t - p_{t-1}}{p_{t-1}}\right) \times 100$$

3. Assume we are interested in the probability that a normal random variable with mean 5 and standard deviation 6 is greater than 6.

We will make use of the *Monte Carlo* (Robert and Casella, 2005) method below. It is a technique to approximate expectations and probabilities. If n is a large number, then the right hand side of

$$\mathbb{P}(X > 6) \approx \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(X_i > 6)$$

is an accurate approximation. If you haven't seen an **indicator** function before, it is defined as

$$\mathbf{1}(X_i > 6) = \begin{cases} 1 & X_i > 6 \\ 0 & X_i \le 6 \end{cases}.$$

a. Evaluate this probability exactly in R and assign it to the variable `exactExceedanceProb`

b. Evaluate this probability exactly in Python and assign it to the variable `exact_exceedance_prob`

c. In R, use the Monte Carlo method to estimate the probability. Use one thousand samples. Assign it to the variable `approxExceedanceProb`

d. In Python, use the Monte Carlo method to estimate the probability. Use one thousand samples. Assign it to the variable `approx_exceedance_prob`

4. For a collection of random variables $X_1, \ldots, X_n$, a *covariance matrix* arranges all of the covariances between every possible pair of random variables:

$$\begin{bmatrix} \text{Cov}(X_1, X_1) & \text{Cov}(X_1, X_2) & \cdots & \text{Cov}(X_1, X_n) \\ \text{Cov}(X_2, X_1) & \text{Cov}(X_2, X_2) & \cdots & \text{Cov}(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(X_n, X_1) & \text{Cov}(X_n, X_2) & \cdots & \text{Cov}(X_n, X_n) \end{bmatrix}$$

where

$$\text{Cov}(X_i, X_j) = \mathbb{E}\left[(X_i - \mathbb{E}[X_i])((X_j - \mathbb{E}[X_j])\right]$$

is the covariance between X_i and X_j resting in row i and column j.

Using this definition, it is easy to show that $\text{Cov}(X_i, X_i) = \mathbb{E}\left[(X_i - \mathbb{E}[X_i])^2\right] = \text{Var}(X_i)$.

An **exchangeable** covariance matrix for a random vector is one that has all the same variances, and all the same covariances. In other words, it has two unique elements: the diagonal elements should be the same, and the off-diagonals should be the same.

a. In R, generate 10 4×4 exchangeable covariance matrices, each with 2 as the variance, and have the possible covariances take values in the collection $0, .01, .02, ..., .09$. Store these 10 covariance matrices in a three-dimensional array. The first index should be each matrix's row index, the second should be the column index of each matrix, and the third index should be the "layer" or "slice" indicating which of the 10 matrices you have. Name this array `myCovMats`

b. Do the same thing in Python, but call the variable `my_cov_mats`

5. In R, read in the `cars.csv` data set using `read.table()` (more on IO in chapter TODO). Find the average `EngineSize`, `Cylinders`, `Horsepower`, `MPG_City`, `MPG_Highway`, `Weight`, `Wheelbase` and `Length` **for each type of vehicle** (i.e. `Hybrid` `Sedan` `Sports`, `SUV`, `Truck` and `Wagon`). Which of these averages is an `NA`? How many observations in that column are missing?

6. In Python (TODO finish this question), read in the `cars.csv` data set using `read.table()` (more on IO in chapter TODO). Find the average `EngineSize`, `Cylinders`, `Horsepower`, `MPG_City`, `MPG_Highway`, `Weight`, `Wheelbase` and `Length` **for each type of vehicle** (i.e. `Hybrid` `Sedan` `Sports`, `SUV`, `Truck` and `Wagon`). Which of these averages is an `NA`? How many observations in that column are missing?

7. Here are two lists in R:

```r
l1 <- list(first="a", second=1)
l2 <- list(first=c(1,2,3), second = "statistics")
```

a. Make a new `list` that is these two lists above "squished together." It has to be length 4, and each element is one of the elements of $l1$ and $l2$. Call this list `l3`.

b. Delete all the "tags" or "names" of these four elements.

c. Make a `vector` of all the unique single digit numbers in both of the lists. You should end up with the vector with elements `1`, `2`, and `3`.

7. Here are two `dicts` in Python:

```
d1 = { "first" : "a", "second" : 1}
d2 = { "first" : [1,2,3], "second" : "statistics"}
```

a. Make a new `list` that is these two `dict`s above "squished together" (why can't it be another `dict`?) It has to be length 4, and each value is one of the values of $d1$ and $d2$. Call this list `l3`.

8. How might you explain the difference between Python and R's type systems? What do you know about the historical development of these languages that might assist your explanation?
9. Example on underflow and overflow.
10. infix functions in R
11. setter methods in R
12. Python functions that have return types specified
13. log-sum-exp trick problem
14. linear algebra problems
15. demo complete.cases versus dropna
16. question on likert scales
17. question with cut that induces NAs

Chapter 3

Odds and Ends

3.1 Input and Output

So far we have been creating small pieces of data within our scripts. This is primarily for pedagogical purposes. In real life, we can have

- data read in from a data set saved on our machine's hard drive (e.g. `my_data.csv`),
- data read in from a data base (e.g. MySQL, PostgreSQL, etc.), or
- data created in a script (either deterministic or random).

I focus mostly on the first one. The third is handled with basic assignment, and using specialized functions that are easily found. I avoid the second one because it requires my setting up a data base and teaching you SQL commands.

After we have created something useful, we might be interested in storing our results. We can write out to a database, a text file, or we can save a digitized version of our work space.

3.1.1 Reading In Text Files

3.2 Using Third-Party Code

Before using third-party code, it must first be installed. After it is installed, it must be "loaded in" to your session. I will describe both of these steps in R and Python.

3.2.1 Installing Packages In R

In R, there are thousands of user-created **packages.** You can download most of these from the *Comprehensive R Archive Network*. You can also download packages from other publishing platforms like Bioconductor, or Github. Installing from CRAN is more commonplace, and extremely easy to do. Just use the `install.packages()` function. This can be run inside your R console, so there is no need to type things into the command line.

```
install.packages("thePackage")
```

3.2.2 Installing Packages In Python

In Python, installing packages is more complicated. Commands must be written in the command line, and there are multiple package managers. This isn't surprising, because Python is used more extensively than R in fields other than data science.

If you followed the suggestions provided in , then you installed Anaconda. This means you will usually be using the `conda` command. Point-and-click interfaces are made available as well.

```
conda install the_package
```

There are some packages that will not be available using this method. For more information on that situation, see here.

3.2.3 Loading Packages In R

After they are installed on your machine, third-party code will need to be "loaded" into your R or Python session.

Loading in a package is relatively simple in R, however complications can arise when different variables share the same name. This happens relatively often because

- it's easy to create a variable in the global environment that has the same name as another object you don't know about, and
- different packages you load in sometimes share names accidentally.

Starting off with the basics, here's how to load in a package of third-party code. Just type the following into your R console.

```
library(thePackage)
```

You can also use the `require()` function, which has slightly different behavior when the requested package is not found.

To understand this more deeply, we need to talk about **environments** again. We discussed these before in 2.5.3, but only in the context of user-defined functions. When we load in a package with `library()`, we make its contents available by putting it all in an environment for that package.

An environment holds the names of objects. There are usually several environments, and each holds a different set of functions and variables. All the variables you define are in an environment, every package you load in gets its own environment, and all the functions that come in R pre-loaded have their own environment.

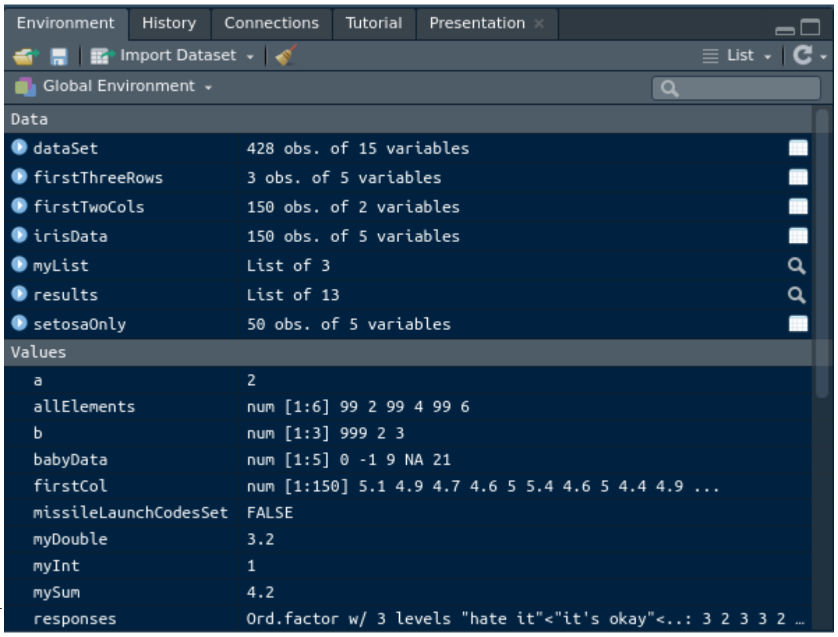
Formally, each environment is pair of two things: a **frame** and an **enclosure**. The frame is the set of symbol-value pairs, and the enclosure is a pointer to the parent environment. If you've heard of a *linked list* in a computer science class, it's the same thing.

Moreover, all of these environments are connected in a chain-like structure. To see what environments are loaded on your machine, and what order they were loaded in, use the `search()` function. This displays the **search path**, or the ordered sequence of all of your environments.

```
search()
##  [1] ".GlobalEnv"         "package:R6"
##  [3] "package:Matrix"     "package:Hmisc"
##  [5] "package:ggplot2"    "package:Formula"
##  [7] "package:survival"   "package:lattice"
##  [9] "package:reticulate" "tools:rstudio"
## [11] "package:stats"      "package:graphics"
## [13] "package:grDevices"  "package:utils"
## [15] "package:datasets"   "package:methods"
## [17] "Autoloads"          "package:base"
```

Alternatively, if you're using RStudio, the search path, and the contents of each of its environments, are displayed in the "Environment" window. You can choose which environment you'd like to look at by selecting it from the dropdown menu. This allows you to see all of the variables in that particular environment. The **global environment** (i.e. `".GlobalEnv"`) is displayed by default, because that is where you store all the objects you are creating in the console.

\begin{figure}

{

```
Data
dataSet                 428 obs. of 15 variables
firstThreeRows          3 obs. of 5 variables
firstTwoCols            150 obs. of 2 variables
irisData                150 obs. of 5 variables
myList                  List of 3
results                 List of 13
setosaOnly              50 obs. of 5 variables
Values
a                       2
allElements             num [1:6] 99 2 99 4 99 6
b                       num [1:3] 999 2 3
babyData                num [1:5] 0 -1 9 NA 21
firstCol                num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
missileLaunchCodesSet   FALSE
myDouble                3.2
myInt                   1
mySum                   4.2
responses               Ord.factor w/ 3 levels "hate it"<"it's okay"<..: 3 2 3 3 2 …
```

}

The Environment Window in RStudio

(#fig:rstudio_disp) \end{figure}

When you call `library(thePackage)`, the package has an environment created for it, and it is *inserted between the global environment, and the most recently loaded package.* When you want to access an object by name, R will first search the global environment, and then it will traverse the environments in the search path in order. These has a few important implications.

- First, **don't define variables in the global environment that are already named in another environment.** There are many variables that come pre-loaded in the `base` package (to see them, type `ls("package:base")`), and if you like using a lot of packages, you're increasing the number of names you should avoid using.

- Second, **don't `library` in a package unless you need it, and if you do, be aware of all the names it will mask it packages you loaded in before**. The good news is that `library` will often print warnings letting you know which names have been masked. The bad news is that it's somewhat out of your control–if you need two packages, then they might have a shared name, and the only thing you can do about it is watch the ordering you load them in.

- Third, don't use `library()` inside code that is `source`'d in other files. For example, if you attach a package to the search path from within a function you defined, anybody that uses your function loses control over the order of packages that get attached.

All is not lost if there is a name conflict. The variables haven't disappeared. It's just slightly more difficult to refer to them. For instance, if I load in `Hmisc`, I get the warning warning that `format.pval` and `units` are now masked because they were names that were in `"package:base"`. I can still refer to these masked variables with the double colon operator (`::`).

```
library(Hmisc)
# format.pval refers to Hmisc's format.pval because it was loaded more recently
# Hmisc::format.pval in this case is the same as above
# base::format.pval this is the only way you can get base's format.pval function
```

3.2.4 Loading Packages In Python

In Python, you use the `import` statement to access objects defined in another file. It is slightly more complicated than R's `library()` function, but it is also more flexible. To make the contents of a package called, say, `the_package` available, type *one of the following* inside a Python session.

```
import the_package
import the_package as tp
from the_package import *
```

To describe the difference between these three approaches, as well as to highlight the important takeaways and compare them with the important takeaways in the last section, we need to discuss what a Python module is, what a package is, and what a Python namespace is.[1]

- A Python **`module`** is a separate (when I say separate, I mean separate from the script file you're currently editing) `.py` file with function and/or object definitions in it.[2]
- A package is a group of modules.[3]
- A **namespace** is "a mapping from names to objects."

With these definitions, we can define `import`ing. According to the Python documentation, "[t]he import statement combines two operations; it searches for the named module, then it binds the results of that search to a name in the local scope."

The sequence of places Python looks for a module is called the search path. This is not the same as R's search path, though. In Python, the search

[1]I am avoiding any mention of *R's* namespaces and modules. These are things that exist, but they are different from Python's namespaces and modules, and are not within the scope of this text.

[2]The scripts you write are modules. They come with the intention of being run from start to finish. Other non-script modules are just a bag of definitions to be used in other places.

[3]Sometimes a package is called a *library* but I will avoid this terminology.

path is a list of places to look for *modules*, not a list of places to look for variables. To see it, `import sys`, then type `sys.path`.

After a module is found, the variable names in the found module become available in the `import`ing module. These variables are available in the global scope, but the names you use to access them will depend on what kind of `import` statement you used. From there, you are using the same scoping rules that we described in 2.5.6, which means the LEGB acronym still applies.

Here are a few important takeaways that might not be readily apparent:

- Python namespaces are unlike R environments in that they are not arranged into a sorted list.
- Unlike in R, there is no *masking*, and you don't have to worry about the *order* of `import`ing things.
- However, you do have to worry about *how* you're `import`ing things. If you use the `from the_package import thingone, thingtwo` format of `import`ing, you are at risk of re-assigning either `thingone` or `thingtwo`, if they already exist. As a rule of thumb, **you should never use this form of `import`ing**.
- These differences might explain why Python packages tend to be larger than R packages.

3.2.4.1 `import`ing Examples

In the example below, we import the entire `numpy` package in a way that lets us refer to it as `np`. This reduces the amount of typing that is required of us, but it also protects against variable name clashing. We then use the `normal()` function to simulate normal random variables. This function is in the `random` sub-module, which is a sub-module in `numpy` that collects all of the pseudorandom number generation functionality together.

```
import numpy as np # import all of numpy
np.random.normal(size=10)
## array([ 1.08748143, -0.64640103,  0.22318977,  1.00367647,  0.04511316,
##        -0.05471008,  0.75755289,  0.10454687,  0.90846735, -1.53474124])
```

This is one use of the dot operator (`.`). It is also used to access attributes and methods of objects (more information on that will come later in chapter 4). `normal` is *inside of* `random`, which it itself inside of `np`.

As a second example, suppose we were interested in the `stats` sub-module found inside the `scipy` package. We could import all of `scipy`, but just like the above example, that would mean we would need to consistently refer to a variable's module, the sub-module, and the variable name. For long programs, this can become tedious if we had to type `scipy.stats.norm` over and over again. Instead, let's import the sub-module (or sub-package) and ignore the rest of `scipy`.

```
from scipy import stats
stats.norm().rvs(size=10)
## array([ 0.31976268,  0.53965409,  1.30604808,  0.67230855,  1.73592725,
##        -0.36731178, -0.21138813, -0.4696466 ,  1.23282603,  0.32604724])
```

So we don't have to type `scipy` every time we use something in `scipy.stats`.

Finally, we can import the function directly, and refer to it with only one letter. This is highly discouraged, though. We are much more likely to accidentally use the name `n` twice. Further, `n` is not a very descriptive name, which means it could be difficult to understand what your program is doing later.

```
from numpy.random import normal as n
n(size=10)
## array([-1.58046049,  2.11855478,  0.49616614,  0.27403613,  0.73194424,
##        -2.13566717,  0.5461117 , -0.15372969,  0.76773148, -0.76083859])
```

Keep in mind, you're always at risk of accidentally re-using names, even if you aren't `import`ing anything. For example, consider the following code.

```
# don't do this!
sum = 3
```

This is very bad, because now you cannot use the `sum` function that was named in the built-in module. To see what is in your built in module, type the following into your Python interpreter: `dir(__builtins__)`.

3.3 Control Flow

3.3.1 Conditional Logic

We discussed Boolean objects in 2.1. We used these for

- counting up number of times a condition appeared, and
- subsetting.

Another way to use them is to conditionally execute code, depending on whether or truth condition of a Boolean.

In R,

```
myName <- "Clare"
if(myName != "Taylor"){
    print("you are not Taylor")
}
## [1] "you are not Taylor"
```

In Python, you don't need curly braces, but the indentation needs to be just right, and you need a colon.

```
my_name = "Taylor"
if my_name == "Taylor":
    print("hi Taylor")
## hi Taylor
```

There can be more than one truth test. To test alternative Boolean conditions, you can add one or more `else if` (in R) or `elif` (in Python) blocks. The first block with a Boolean that is found to be true will execute, and none of the resulting conditions will be checked.

If no `if` block or `else if`/`elif` block executes, an `else` block will always execute. That's why `else` blocks don't need to look at a Boolean. Whether they execute only depends on the Booleans in the previous blocks.

```
food <- "muffin"
if(food == "apple"){
    print("an apple a day keeps the doctor away")
}else if(food == "muffin"){
    print("muffins have a lot of sugar in them")
}else{
    print("neither an apple nor a muffin")
}
## [1] "muffins have a lot of sugar in them"
```

```
my_num = 42.999
if my_num % 2 == 0:
    print("my_num is even")
elif my_num % 2 == 1:
    my_num += 1
    print("my_num was made even")
else:
    print("you're cheating by not using integers!")
## you're cheating by not using integers!
```

3.3.2 Loops

One line of code generally does one "thing," unless you're using loops. Code written inside a loop will execute many times.

The most common loop for us will be a `for` loop. A simple `for` loop in R might look like this

```
myLength <- 9
r <- vector(mode = "numeric", length = myLength)
for(i in seq_len(myLength)){
    r[i] <- i
}
r
## [1] 1 2 3 4 5 6 7 8 9
```

1. `seq_len(myLength)` gives us a `vector`

2. `i` is a variable that takes on the values found in the `vector`
3. Code inside the loop (inside the curly braces), is repeatedly executed, and it may or may not reference the dynamic variable `i`

In Python

```
my_length = 9
r = []
for i in range(my_length):
    r.append(i)
r
## [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

1. Unsurprisingly, Python's syntax opts for indentation and colons instead of curly braces and parentheses,
2. Code inside the loop (inside the curly braces), is repeatedly executed, and it may or may not reference the dynamic variable `i`
3. `for` loops in Python are more flexible because they iterate over many different types of data structures,
4. The `range` doesn't generate all the numbers in the sequence at once, so it saves on memory. This can be quite useful for certain applications. However, `r` is a list that *does* store all the consecutive integers.

Loop tips:

.enumienumi.If you find yourself copy/paste-ing code, changing only a small portion of text on each line of code, you should consider using a loop, If a `for` loop works for something you are trying to do, first try to find a replacement function that does what you want. The examples above just made a `vector`/`list` of consecutive integers. There are many built in functions that accomplish this. Avoiding loops in this case would make your program shorter, easier to read, and (potentially) much faster. A third option between looping, and a built-in function, is to try the functional approach. This will be explained more in the last chapter. Watch out for **off-by-one** errors. Iterating over the wrong sequence is a common mistake, considering

1. • Python starts counting from 0, while R starts counting from 1

  - sometimes iteration `i` references the `i-1`th element of a container
  - The behavior of loops is sometimes more difficult to understand if they're using `break` or `continue`/`next` statements
5. *Don't hardcode variables.* Minimize the number of places you have to make changes to your code. You will change your code consistently, so save your future self some time.

The last point bears repeating: *don't hardcode variables*. In statistical programs, there are often "tuning parameters," for instance that must be changed frequently to affect the overall behavior of the program. If these variables only need to be changed in one location, that saves you a lot of time and gives you more flexibility.

In the example above, the `myLength` or `my_length` variable could be referenced in many places throughout the entire program. If you wanted to change the number of iterations in your program (which happens all the time), and you *did* hardcode the length in a bunch of places throughout the program, you would need to hunt down all those changes!

Python provides an alternative way to construct lists similar to the one we constructed in the above example. They are called **list comprehensions**. You can incorporate iteration and conditional logic in one line of code.

```
[3*i for i in range(10) if i%2 == 0]
## [0, 6, 12, 18, 24]
```

You might also have a look at *generator expressions* and *dictionary comprehensions*.

R can come close to replicating the above behavior with vectorization, but the conditional part is hard to achieve without subsetting.

```
3*seq(0,9)[seq(0,9)%%2 == 0]
## [1]  0  6 12 18 24
```

3.3.3 A Longer Example

3.3.3.1 Description of Accept-Reject Sampling

An example of an algorithm that uses conditional logic is the **accept-reject sampling method** (Robert and Casella, 2005). This is useful for when we want to sample from a *target probability density* $p(x)$, using another distribution called a *proposal ditribution* $q(x)$.

$q(x)$ is probably a distribution that is easy to sample from and is easy to evaluate pointwise. For example, a uniform distribution satisfies these criteria because both R and Python have functions that accomplish these two things (e.g. sampling can be done with `runif` in R and `np.random.uniform` in Python). $p(x)$ is generally more "complicated." If it wasn't, we would try to find some built-in function for it.

One common way a distribution can be complicated is that it can have an unknown **normalizing constant**–one that is difficult or impossible to solve using calculus. This happens a lot in *Bayesian Statistics*, for example.[4]. We might write down

$$p(x) = \frac{f(x)}{\int f(x)dx},$$

and this is guaranteed to be a probability density function as long as $f(x) \geq 0$ and $\int f(x)dx < \infty$, but we might have no idea how to solve the denominator. In this case, $f(x)$ is easy to evaluate pointwise, but $p(x)$ is not.

This algorithm makes use of an auxiliary random variable that is sampled from a Bernoulli(p) distribution. As long as $0 < p < 1$, a Bernoulli random variable Y is either 0 or 1. The probability it takes the value 1 is p, while the probability that it takes the value 0 is $1 - p$. A coin flip is a good example use-case for this distribution. Coin flips are commonly assumed to be distributed asBernoulli(.5). At least for fair coins, there is an equal chance that the coin lands heads (i.e. 0) or tails (i.e. 1).

The most difficult part about using this algorithm is that one must calculate the probability parameter of this Bernoulli random variable. This involves calculating (by hand) an upper bound M for the ratio $f(x)/q(x)$.

[4]The posterior distribution is usually the object of interest in Bayesian statistics. According to Bayes' Rule, the unnormalized posterior is usually the product of two "easy" functions. However, integrating the product is not always possible!

This bound has to hold uniformly, meaning that it is a constant number that is greater than the ratio no matter what x we plug in.

Below is one step of the accept-reject algorithm.

Algorithm 1: Accept-Reject Sampling (One Step)

---

1. Calculate $M > \frac{f(x)}{q(x)}$ (the smaller the better)
2. Sample X from $q(x)$
3. Sample $Y \mid X$ from Bernoulli $\left(\frac{f(X)}{q(X)M}\right)$
4. If $Y = 1$, then return X
5. Otherwise, return nothing

Multiple samples will be required, so this process needs to be iterated many times. There are two ways to do this. If you want to iterate a fixed number of times, you can use a `for` loop. However, in that case, you will end up with a random number of samples. On the other hand, if you want a nonrandom number of samples, you will probably want a `while` loop. This is the approach the example below takes. The `while` loop will continue iterating until a condition is false. In our case, we want to loop until we receive the total number of samples we requested.

3.3.3.2 A Specific Example

Here is a specific example. Let's say our target[5] is

$$p(x) = \begin{cases} \frac{x^2(1-x)}{\int_0^1 x^2(1-x)dx} & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}.$$

[5]This is the density of a Beta$(3, 2)$ random variable, if you're curious.

The denominator, $\int_0^1 x^2(1-x)dx$, is the target's normalizing constant. You might know how to solve this integral, but let's pretend for the sake of our example that it's too difficult for us. We want to sample from $p(x)$ while only being able to evaluate (not sample) from its normalized version.

Next, let's choose a uniform distribution for our proposal distribution:

$$q(x) = \begin{cases} 1 & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

We can plot all three functions.

Here's some Python code that attempts to sample once from $p(x)$. Sometimes proposals are not accepted. When that happens, the function returns `None`.

```python
import numpy as np

def f(samp):
    """the unnormalized density"""
    return (1-samp)*(samp**2)

def attempt_one_samp():
    """attempts to sample from target distribution, using uniform as a proposal"""
    x = np.random.uniform()
    M = 4/27
    bern_prob_param = f(x)/M
    accept = np.random.binomial(1, bern_prob_param) == 1
    if accept:
        return x
```

```python
def sample_from_target(num_times):
    """sample num_times from the target distribution"""
    samps = []
    while len(samps) < num_times:
        potential_samp = attempt_one_samp()
        if potential_samp is not None:
            samps.append(potential_samp)
    return samps
```

1. we used a `while` loop instead of a `for` loop because we did not know how many iterations it would take to get `num_times` samples
2. We are following the Python style guide and using the `is not` keyword to check if something is `None`

In chapter 3.5, we'll show you the code that you can use to generate the plot below. `r_and_python_book_files/figure-latex/unnamed-ch`

3.4 Reshaping and Combining Data Sets

3.4.1 Ordering and Sorting Data

Sorting a data set, in ascending order, say, is a common task. You might need to do it because

1. ordering and ranking is commonly done in *nonparametric statistics*,
2. you want to inspect the most "extreme" observations in a data set,
3. it's a pre-processing step before generating visualizations.

In R, it all starts with `vectors`. There are two common functions you should know: `sort` and `order`. `sort` returns the sorted *data*, while `order` returns the *order indexes*.

```
sillyData <- rnorm(5)
print(sillyData)
## [1] -0.1743 -0.8414  0.7660 -0.2070 -1.2058
sort(sillyData)
```

```
## [1] -1.2058 -0.8414 -0.2070 -0.1743  0.7660
order(sillyData)
## [1] 5 2 4 1 3
```

`order` is useful if you're sorting a data frame by a particularly column. Below, we inspect the top 5 most expensive cars. Notice that we need to clean up the `MSRP` (a `character` vector) a little first. We use the function `gsub` to find patterns in the text, and replace them with the empty string.

```
carData <- read.csv("data/cars.csv")
noDollarSignMSRP <- gsub("$", "", carData$MSRP, fixed = TRUE)
carData$cleanMSRP <- as.numeric(gsub(",", "", noDollarSignMSRP, fixed = TRUE))
rowIndices <- order(carData$cleanMSRP, decreasing = TRUE)[1:5]
carData[rowIndices,c("Make", "Model", "MSRP", "cleanMSRP")]
```

```
##              Make                    Model     MSRP
## 335       Porsche            911 GT2 2dr $192,465
## 263 Mercedes-Benz              CL600 2dr $128,420
## 272 Mercedes-Benz SL600 convertible 2dr $126,670
## 271 Mercedes-Benz           SL55 AMG 2dr $121,770
## 262 Mercedes-Benz              CL500 2dr  $94,820
##     cleanMSRP
## 335    192465
## 263    128420
## 272    126670
## 271    121770
## 262     94820
```

In Python, Numpy has `np.argsort` and `np.sort`.

```
import numpy as np
silly_data = np.random.normal(size=5)
print(silly_data)
## [ 0.37711014 -0.04447526 -0.0704097   0.42889182  2.42969921]
np.sort(silly_data)
## array([-0.0704097 , -0.04447526,  0.37711014,  0.42889182,  2.42969921])
```

```
np.argsort(silly_data)
## array([2, 1, 0, 3, 4])
```

For Pandas' `DataFrames`, most of the functions I find useful are methods attached to the `DataFrame` class. That means that, as long as something is inside a `DataFrame`, you can use dot notation.

```
import pandas as pd
car_data = pd.read_csv("data/cars.csv")
car_data['no_dlr_msrp'] = car_data['MSRP'].str.replace("$", "", regex = False)
car_data['clean_MSRP'] = car_data['no_dlr_msrp'].str.replace(",","").astype(fl
car_data = car_data.sort_values(by='clean_MSRP', ascending = False)
car_data[["Make", "Model", "MSRP", "clean_MSRP"]].head(5)
##               Make                    Model      MSRP  clean_MSRP
## 334        Porsche              911 GT2 2dr  $192,465    192465.0
## 262  Mercedes-Benz                CL600 2dr  $128,420    128420.0
## 271  Mercedes-Benz  SL600 convertible 2dr  $126,670    126670.0
## 270  Mercedes-Benz              SL55 AMG 2dr  $121,770    121770.0
## 261  Mercedes-Benz                CL500 2dr   $94,820     94820.0
```

Pandas `DataFrames` and `Series` have a `replace` method. We use this to remove dollar signs and commas from the MSRP column. Note that we had to access the `.str` attribute of the `Series` column before we used it. After the string was processed, we converted it to a `Series` of `floats` with the `astype` method.

Finally, sorting the overall data frame could have been done with the same approach as the code we used in R (i.e. raw subsetting by row indexes), but there is a built in method called `sort_values` that will do it for us.

3.4.2 Stacking Data Sets and Placing them Shoulder to Shoulder

Stacking data sets on top of each other is a common task. You might need to do it if

1. you need to add new a new row (or many rows) to a data frame,

2. you need to recombine data sets (e.g. recombine a train/test split), or
3. you're creating a matrix in a step-by-step way.

In R, this can be done with `rbind` (short for "row bind")

```
realEstate <- read.csv("data/albemarle_real_estate.csv")
train <- realEstate[-1,]
test <- realEstate[1,]
head(rbind(test, train))
##   YearBuilt YearRemodeled Condition NumStories FinSqFt
## 1      2006             0   Average       1.00    1922
## 2      2003             0   Average       1.00    1848
## 3      1972             0   Average       1.00    1248
## 4      1998             0      Good       1.00    1244
## 5      1886             0   Average       1.86    1861
## 6      1910             0      Fair       1.53    1108
##   Bedroom FullBath HalfBath TotalRooms LotSize
## 1       3        3        0         10   5.000
## 2       3        2        0          7  61.189
## 3       2        1        0          4   1.760
## 4       1        1        0          3  50.648
## 5       4        1        0          6   3.880
## 6       3        1        0          6   8.838
##   TotalValue        City
## 1     409900      CROZET
## 2     523100      CROZET
## 3     180900 EARLYSVILLE
## 4     620700      CROZET
## 5     162500      CROZET
## 6     167200      CROZET
sum(rbind(test, train) != realEstate)
## [1] 0
```

The above example was with `data.frames`. This example of `rbind` is with `matrix` objects.

```
rbind(matrix(1,nrow = 2, ncol = 3),
      matrix(2,nrow = 2, ncol = 3))
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    2    2    2
## [4,]    2    2    2
```

In Python, you can stack data frames with `pd.concat`. It has a lot of options, so feel free to peruse those. You can also replace the call to `pd.concat` below with `test.append(train)`.

```
import pandas as pd
real_estate = pd.read_csv("data/albemarle_real_estate.csv")
train = real_estate.iloc[1:,]
test = real_estate.iloc[[0],] # need the extra brackets!
pd.concat([test,train], axis=0).head() # also
##    YearBuilt  YearRemodeled Condition  ...  LotSize  TotalValue         C
## 0       2006              0   Average  ...    5.000      409900       CRO
## 1       2003              0   Average  ...   61.189      523100       CRO
## 2       1972              0   Average  ...    1.760      180900  EARLYSVI
## 3       1998              0      Good  ...   50.648      620700       CRO
## 4       1886              0   Average  ...    3.880      162500       CRO
##
## [5 rows x 12 columns]
(pd.concat([test,train], axis=0) != real_estate).sum().sum()
## 0
```

Take note of the extra square brackets when we create `test`. If you use `real_estate.iloc[0,]` instead, it will return a `Series` with all the elements coerced to the same type, and this won't `pd.concat` properly with the rest of the data!

3.4.3 Merging or Joining Data Sets

If you have two different data sets that provide different information about the same things, you put them together using a **`merge`** (aka **`join`**) statement. The resulting data set is wider, and possibly with fewer rows. In R, you can use the `merge` function. In Python, you can use the `merge` method.

Suppose you have to sets of supposedly anonymized data about individual accounts on some online platforms.

```
baby1 <- read.csv("data/baby1.csv", stringsAsFactors = FALSE)
baby2 <- read.csv("data/baby2.csv", stringsAsFactors = FALSE)
head(baby1)
##   idnum height.inches.          email_address
## 1     1             74 fakeemail123@gmail.com
## 2     3             66  anotherfake@gmail.com
## 3     4             62      notreal@gmail.com
## 4    23             62      notreal@gmail.com
head(baby2)
##     idnum     phone                    email
## 1 3901283 5.051e+09       notreal@gmail.com
## 2   41823 5.051e+09 notrealeither@gmail.com
## 3 7198273 5.051e+09   anotherfake@gmail.com
```

The first thing you need to ask yourself is "which column is the unique identifier that is shared between these two data sets?" In our case, they both have an "identification number" column, could that be it? Let's suppose for the sake of argument that these two data sets are coming from different online platforms, and these two places use different schemes to number their users.

In this case, they both share a column with (possibly) the same information about email addresses. They are named differently in each data set, so we must specify both column names.

```
# in R
merge(baby1, baby2, by.x = "email_address", by.y = "email")
##           email_address idnum.x height.inches. idnum.y
## 1 anotherfake@gmail.com       3             66 7198273
## 2     notreal@gmail.com       4             62 3901283
## 3     notreal@gmail.com      23             62 3901283
##        phone
## 1 5.051e+09
## 2 5.051e+09
## 3 5.051e+09
```

In Python, `merge` is a method attached to each `DataFrame` instance.

```
# in Python
baby1.merge(baby2, left_on = "email_address", right_on = "email")
##    idnum_x  height(inches)  ...       phone                    email
## 0        3              66  ...  5051234568  anotherfake@gmail.com
## 1        4              62  ...  5051234567      notreal@gmail.com
## 2       23              62  ...  5051234567      notreal@gmail.com
##
## [3 rows x 6 columns]
```

The email addresses `anotherfake@gmail.com` and `notreal@gmail.com` exist in both data sets, so each of these email addresses will end up in the result data frame. The rows in the result data set are wider and have more attributes for each individual.

Notice the duplicate email address, too. In this case, either the user signed up for two accounts using the same email, or one person signed up for an account with another person's email address. In the case of duplicates, both rows will match with the same rows in the other data frame.

Also, in this case, all email addresses that weren't found in both data sets were thrown away. This does not necessarily need to be the intended behavior. For instance, if we wanted to make sure no rows were thrown away, that would be possible. In this case, though, for email addresses that weren't found in both data sets, some information will be missing. Recall that Python and R handle missing data differently (see 2.2.8).

```
# in R
merge(baby1, baby2,
      by.x = "email_address", by.y = "email",
      all.x = TRUE, all.y = TRUE)
##             email_address idnum.x height.inches.
## 1   anotherfake@gmail.com       3              66
## 2  fakeemail123@gmail.com       1              74
## 3        notreal@gmail.com       4              62
## 4        notreal@gmail.com      23              62
## 5 notrealeither@gmail.com      NA              NA
##   idnum.y     phone
## 1 7198273 5.051e+09
## 2      NA        NA
## 3 3901283 5.051e+09
## 4 3901283 5.051e+09
## 5   41823 5.051e+09
```

```
# in Python
baby1.merge(baby2,
            left_on = "email_address", right_on = "email",
            how = "outer")
##    idnum_x  height(inches)  ...         phone                    email
## 0      1.0            74.0  ...           NaN                      NaN
## 1      3.0            66.0  ...  5.051235e+09    anotherfake@gmail.com
## 2      4.0            62.0  ...  5.051235e+09        notreal@gmail.com
## 3     23.0            62.0  ...  5.051235e+09        notreal@gmail.com
## 4      NaN             NaN  ...  5.051235e+09  notrealeither@gmail.com
##
## [5 rows x 6 columns]
```

You can see it's slightly more concise in Python. If you are familiar with SQL, you might have heard of inner and outer joins. This is where Pandas takes some of its argument names from.

3.4.4 Long Versus Wide Data

3.4.4.1 Long Versus Wide in R

Many types of data can be stored in either a **wide** or **long** format.

The classical example is data from a *longitudinal study.* If an experimental unit (in the example below a person) is repeatedly measured over time, each row would correspond to an experimental unit *and* an observation time in a data set in a long form.

```
fake_long_data1 <- data.frame(person = c("Taylor","Taylor","Charlie","Charlie"
                              timeObserved = c(1, 2, 1, 2),
                              nums = c(100,101,300,301))
fake_long_data1
##    person timeObserved nums
## 1  Taylor            1  100
## 2  Taylor            2  101
## 3 Charlie            1  300
## 4 Charlie            2  301
```

A long format can also be used if you have multiple observations (at a single time point) on an experimental unit. Here is another example.

```
fake_long_data2 <- data.frame(person = c("Taylor", "Taylor", "Charlie","Charli
                              attributeName = c("attrA","attrB","attrA","attrB"
                              nums = c(100,101,300,301))
fake_long_data2
##    person attributeName nums
## 1  Taylor         attrA  100
## 2  Taylor         attrB  101
## 3 Charlie         attrA  300
## 4 Charlie         attrB  301
```

If you would like to reshape the long data sets into a wide format, you can use the `reshape` function. You will need to specify which columns correspond with the experimental unit, and which column is the "factor" variable.

```
fake_wide_data1 <- reshape(fake_long_data1,
                           direction = "wide",
                           timevar = "timeObserved",
                           idvar = "person",
                           varying = c("before","after")) # col names in new
fake_long_data1
```

```
##    person timeObserved nums
## 1  Taylor            1  100
## 2  Taylor            2  101
## 3 Charlie            1  300
## 4 Charlie            2  301
fake_wide_data1
##    person before after
## 1  Taylor    100   101
## 3 Charlie    300   301
```

```
fake_wide_data2 <- reshape(fake_long_data2,
                           direction = "wide",
                           timevar = "attributeName", # timevar is kind of a misnome
                           idvar = "person",
                           varying = c("attribute A","attribute B"))
fake_long_data2
##    person attributeName nums
## 1  Taylor         attrA  100
## 2  Taylor         attrB  101
## 3 Charlie         attrA  300
## 4 Charlie         attrB  301
fake_wide_data2
##    person attribute A attribute B
## 1  Taylor         100         101
## 3 Charlie         300         301
```

`reshape` will also go in the other direction: it can take wide data and convert it into long data

```
reshape(fake_wide_data1,
        direction = "long",
        idvar = "person",
        varying = list(c("before","after")),
        v.names = "nums")
##            person time nums
## Taylor.1   Taylor    1  100
## Charlie.1 Charlie    1  300
## Taylor.2   Taylor    2  101
## Charlie.2 Charlie    2  301
fake_long_data1
##    person timeObserved nums
## 1  Taylor            1  100
## 2  Taylor            2  101
## 3 Charlie            1  300
## 4 Charlie            2  301
reshape(fake_wide_data2,
        direction = "long",
        idvar = "person",
        varying = list(c("attribute A","attribute B")),
        v.names = "nums")
##             person time nums
## Taylor.1    Taylor    1  100
## Charlie.1 Charlie    1  300
## Taylor.2    Taylor    2  101
## Charlie.2 Charlie    2  301
fake_long_data2
##    person attributeName nums
## 1  Taylor         attrA  100
## 2  Taylor         attrB  101
## 3 Charlie         attrA  300
## 4 Charlie         attrB  301
```

3.4.4.2 Long Versus Wide in Python

With Pandas, we can take make wide data long with `pd.DataFrame.pivot`, and we can go in the other direction with `pd.DataFrame.melt`.

When going from wide to long, make sure to use the `pd.DataFrame.reset_index()` method afterwards to reshape the data and remove the index. Here is an example similar to the one above.

```
import pandas as pd
fake_long_data1 = pd.DataFrame({'person' : ["Taylor","Taylor","Charlie","Charl
                               'time_observed' : [1, 2, 1, 2],
                               'nums' : [100,101,300,301]})
fake_long_data1
##     person  time_observed  nums
## 0   Taylor              1   100
## 1   Taylor              2   101
## 2  Charlie              1   300
## 3  Charlie              2   301
pivot_data1 = fake_long_data1.pivot(index='person', columns='time_observed', values='
pivot_data1
## time_observed    1    2
## person
## Charlie        300  301
## Taylor         100  101
fake_wide_data1 = pivot_data1.reset_index()
fake_wide_data1
## time_observed   person    1    2
## 0              Charlie  300  301
## 1               Taylor  100  101
```

Here's one more example showing the same functionality–going fom wide to long format.

```
fake_long_data2 = pd.DataFrame({'person' : ["Taylor","Taylor","Charlie","Charlie"],
                               'attribute_name' : ['attrA', 'attrB', 'attrA', 'attrB'
                               'nums' : [100,101,300,301]})
fake_wide_data2 = fake_long_data2.pivot(index='person',
                                        columns='attribute_name',
                                        values='nums').reset_index()
fake_wide_data2
## attribute_name   person  attrA  attrB
## 0               Charlie    300    301
## 1                Taylor    100    101
```

Here are some examples of going in the other direction: from wide to long with `pd.DataFrame.melt`. The first example specifies value columns by integers.

```
fake_wide_data1
## time_observed   person    1    2
## 0              Charlie  300  301
## 1               Taylor  100  101
fake_wide_data1.melt(id_vars = "person", value_vars = [1,2])
##     person time_observed  value
## 0  Charlie             1    300
## 1   Taylor             1    100
## 2  Charlie             2    301
## 3   Taylor             2    101
```

The second example uses strings to specify value columns.

```
fake_wide_data2
## attribute_name   person  attrA  attrB
## 0               Charlie    300    301
## 1                Taylor    100    101
fake_wide_data2.melt(id_vars = "person", value_vars = ['attrA','attrB'])
##     person attribute_name  value
## 0  Charlie          attrA    300
## 1   Taylor          attrA    100
## 2  Charlie          attrB    301
## 3   Taylor          attrB    101
```

3.5 Visualization

I describe a few plotting paradigms in R and Python below. Note that these descriptions are extremely thin. More depth could easily turn any of these subsections into an entire textbook!

3.5.1 Base R Plotting

R comes with some built-in functions `plot`, `hist`, `boxplot`, etc. Many of these reside in `package:graphics`, which comes pre-loaded into the search path. `plot` on the other hand, is higher up the search path in package:base–it is a generic method whose methods might be in package:graphics or some place else.

Base plotting covers most needs, so that's what we spend most time with. However, there are a large number of third-party libraries for plotting that you might consider looking into if you want to follow a certain aesthetic, or if you want plotting specialized for certain cases (e.g. geospatial plots).

Recall our Albemarle Real Estate data set.

```
df <- read.csv("data/albemarle_real_estate.csv")
head(df)
##   YearBuilt YearRemodeled Condition NumStories FinSqFt
## 1      2006             0   Average       1.00    1922
## 2      2003             0   Average       1.00    1848
## 3      1972             0   Average       1.00    1248
## 4      1998             0      Good       1.00    1244
## 5      1886             0   Average       1.86    1861
## 6      1910             0      Fair       1.53    1108
##   Bedroom FullBath HalfBath TotalRooms LotSize
## 1       3        3        0         10   5.000
## 2       3        2        0          7  61.189
## 3       2        1        0          4   1.760
## 4       1        1        0          3  50.648
## 5       4        1        0          6   3.880
## 6       3        1        0          6   8.838
##   TotalValue        City
## 1     409900      CROZET
## 2     523100      CROZET
## 3     180900 EARLYSVILLE
## 4     620700      CROZET
## 5     162500      CROZET
## 6     167200      CROZET
```

If we wanted to get a general idea of how expensive homes were in Albemarle County, we could use a histogram. This helps us visualize a univariate numerical variable/column. Below I plot the (natural) logarithm of home prices.

```r
hist(log(df$TotalValue),
     xlab = "natural logarithm of home price", main = "Super-Duper Plot!")
```

r_and_python_book_files/figure-latex/unnamed-chunk-

I specified the `xlab=` and `main=` arguments, but there are many more that could be tweaked. Make sure to skim the options in the documentation (`?hist`).

`plot` is useful for plotting two univariate numerical variables. This can be done in time series plots (variable versus time) and scatter plots (one variable versus another).

```r
par(mfrow=c(1,2))
plot(df$TotalValue, df$LotSize,
     xlab = "total value ($)", ylab = "lot size (sq. ft.)",
     pch = 3, col = "red", type = "b")
plot(log(df$TotalValue), log(df$LotSize),
     xlab = "log. total value", ylab = "log. lot size",
     pch = 2, col = "blue", type = "p")
abline(h = log(mean(df$LotSize)), col = "green")
```

r_and_python_book_files/figure-latex/unnamed-chunk-

```r
par(mfrow=c(1,1))
```

I use some of the many arguments available (type `?plot`). `xlab=` and `ylab=` specify the x- and y-axis labels, respectively. `col=` is short for "color." `pch=` is short for "point character." Changing this will change the symbol shapes used for each point. `type=` is more general than that, but it is related. I typically use it to specify whether or not I want the points connected with lines.

I also use a couple other functions. `abline` is used to superimpose lines over the top of a plot. They can be `h`orizontal, `v`ertical, or you can specify them in slope-intercept form, or by providing a linear model object. I also used `par` to set a graphical parameter. The graphical parameter `par()$mfrow` sets the layout of a multiple plot visualization. I then set it back to the standard 1×1 layout afterwards.

3.5.2 Plotting with `ggplot2`

`ggplot2` is a popular third-party visualization package for R. There are also libraries in Python (e.g. `plotnine`) that look and feel quite similar. This subsection provides a short tutorial on how to use `ggplot2` in R, and it is primarily based off of the material provided in (Wickham, 2016). An online version of this text is provided here.

`ggplot2` code looks a lot different than the code in the above section[6]. There, we would write a series of function calls, and each would change some state in the current figure. Here, we call different `ggplot2` functions that create S3 objects with special behavior (more information about S3 objects in subsection 4.2.2), and then we "add" (i.e. we use the `+` operator) them together.

This new design is not to encourage you to think about S3 object-oriented systems. Rather, it is to get you thinking about making visualizations using the "grammar of graphics" (Wilkinson, 2005). `ggplot2` makes use of its own specialized vocabulary that is taken from this book. As we get started, I will try to introduce some of this vocabulary slowly.

The core function is the `ggplot` function. This is the function that figures are initialized with; it is the function that will take in information about *which* data set you want to plot, and *how* you want to plot it. The raw data is provided in the first argument. The second argument, `mapping=`, is

[6]Personally, I find its syntax more confusing, and so I tend to prefer base graphics. However, it is popular at the moment, and so I do believe that it is important to mention here in this text.

more confusing. The argument should be constructed with the `aes` function. In the parlance of `ggplot2`, `aes` constructs an **aesthetic mapping**. Think of the "aesthetic mapping" as stored information that can be used later on–it "maps" data to visual properties of a figure.

Consider this first example.

```
library(ggplot2)
ggplot(mpg, aes(x = displ, y = hwy))
```

You'll notice a few things about the code and the result produced:

1. No geometric shapes show up!
2. A Cartesian coordinate system is displayed, and the x-axis and y-axis were created based on aesthetic mapping provided (confirm this by typing `summary(mpg$displ)` and `summary(mpg$hwy)`).
3. The axis labels are taken from the column names provided to `aes`.

To plot geometric shapes (*geoms* in the parlance of `ggplot2`), we need to add **layers** to it. "Layers" is quite a broad term–it does not only apply to geometric objects. In fact, in `ggplot2`, a layer can be pretty much anything: raw data, summarized data, transformed data, annotations, etc. However, the functions that add geometric object layers usually start with the prefix `geom_`. In RStudio, after loading `ggplot2`, type `geom_`, and then press `<Tab>` (autocomplete) to see some of the options.

Consider the function `geom_point`. It too returns an S3 instance that has specialized behavior. In the parlance of `ggplot2`, it adds a scatterplot layer to the figure.

```
library(ggplot2)
ggplot(mpg, aes(x = displ, y = hwy))  +
  geom_point()
```

r_and_python_book_files/figure-latex/unnamed-chunk-157-1.pd

Notice that we did not need to provide any arguments to `geom_point`. The aesthetic mappings were used by the new layer.

There are *many* types of layers that you can add to a plot, and you're not limited to any number of them in a given plot. For example, if we wanted to add a title, we could use the `ggtitle` function to add a title layer. Unlike `geom_point`, this function will need to take an argument because the desired title is not stored as an aesthetic mapping.

```
ggplot(mpg, aes(x = displ, y = hwy))  +
  geom_point() +
  ggtitle("my favorite scatterplot")
```

r_and_python_book_files/figure-latex/unnamed-chunk-158-1.pd

Additionally, notice that the same layer will behave much differently if we change the aesthetic mapping.

```
ggplot(mpg, aes(x = displ, y = hwy, color = manufacturer))  +
  geom_point() +
  ggtitle("my favorite scatterplot")
```

r_and_python_book_files/figure-latex/unnamed-chunk-159-1.pd

If we want tighter control on the aesthetic mapping, we can use **scales**. Syntactically, these are things we "add" (`+`) to the figure, just like layers.

However, these scales are constructed with a different set of functions, many of which start with the prefix `scale_`.

We can change attributes of the axes like this.

```r
base_plot <- ggplot(mpg, aes(x = displ, y = hwy, color = manufacturer))  +
              geom_point() +
              ggtitle("my favorite scatterplot")
base_plot + scale_x_log10() + scale_y_reverse()
```

r_and_python_book_files/figure-latex/unnamed-chunk-1

We can also change plot colors with scale layers. Let's add an aesthetic called fill so we can use colors to denote the value of a numerical (not categorical) column. This data set doesn't have any more unused numerical columns, so let's create a new one called `score`. We also use a new geom layer from a function called `geom_tile()`.

```r
mpg$score <- 1/(mpg$displ^2 + mpg$hwy^2)
ggplot(mpg, aes(x = displ, y = hwy, fill = score ))  +
  geom_tile()
```

r_and_python_book_files/figure-latex/unnamed-chunk-1

If we didn't like these colors, we could change them with a scale layer. Personally, I like this one.

```r
mpg$score <- 1/(mpg$displ^2 + mpg$hwy^2)
ggplot(mpg, aes(x = displ, y = hwy, fill = score ))  +
  geom_tile() +
  scale_fill_viridis_b()
```

r_and_python_book_files/figure-latex/unnamed-chunk-162-1.pd

There are many to choose from, though. Here's another one.

```
mpg$score <- 1/(mpg$displ^2 + mpg$hwy^2)
ggplot(mpg, aes(x = displ, y = hwy, fill = score ))  +
  geom_tile() +
  scale_fill_gradient2()
```

r_and_python_book_files/figure-latex/unnamed-chunk-163-1.pd

3.5.3 Plotting with Matplotlib

Matplotlib (Hunter, 2007) is a third-party visualization library in Python. It's the oldest and most heavily-used, so it's the best way to start making graphics in Python, in my humble opinion. It also comes installed with Anaconda. This short introduction borrows heavily from the myriad of tutorials on Matplotlib's website. I will start off making a simple plot, and commenting on each line of code.

You can use either "pyplot-style" (e.g. `plt.plot()`) or "object-oriented-style" to make figures in Matplotlib. Even though using the first type is faster to make simple plots, I will only describe the second one. It is the recommended approach because it is more extensible. However, the first one resembles the syntax of MATLAB. If you're familiar with MATLAB, you might consider learning a little about the first style, as well.

```
import matplotlib.pyplot as plt       # 1
import numpy as np                    # 2
fig, ax = plt.subplots()              # 3
ax.hist(np.random.normal(size=1000))  # 4
## (array([  6.,   15.,   76., 177., 259., 243., 158.,   53.,   11.,   2.]), arr
##         0.06090982,  0.77385815,  1.48680647,  2.19975479,  2.91270311,
##         3.62565144]), <a list of 10 Patch objects>)
```

In the first line, we import the `pyplot` submodule of `matplotlib`. We rename it to `plt`, which is short, and will save us some typing. It also follows the most commonly-used convention.

Second, we import Numpy in the same way we always have. Matplotlib is written to work with Numpy arrays. If you want to plot some data, and it isn't in a Numpy array, you should convert it first.

Third, we call the `subplots` function, and use *sequence unpacking* to unpack the returned container into individual objects without storing the overall container. "Subplots" sounds like it will make many different plots all on one figure, but if you look at the documentation the number of rows and columns defaults to one and one, respectively.

`plt.subplots` returns a `tuple`[7] of two things: a `Figure` object, and one or more `Axes` object(s). These two classes will require some explanation.

1. A `Figure` object is the overall visualization object you're making. It holds onto all of the plot elements. If you want to save all of your progress (e.g. with `fig.savefig('my_picture.png')`), you're saving the overall `Figure` object.

2. One or more `Axes` objects are contained in a `Figure` object. Each is "what you think of as 'a plot'." They hold onto two `Axis` objects (in the case of 2-dimensional plots) or three (in the case of 3-dimensional

[7]We didn't talk about `tuples` in chapter 2, but you can think of them as being similar to `lists`. They are containers that can hold elements of different types. There are a few key differences, though: they are made with parentheses (e.g. `('a')`) instead of square brackets, and they are immutable instead of mutable.

arguments). We are usually calling the methods of these objects to effect changes on a plot.

In line four, we call the `hist()` method of the `Axes` object called `ax`. There are many more plots available than plain histograms. Each one has its own method, and you can peruse the options in the documentation.

If you want to make figures more elaborate, just keep calling different methods of `ax`. If you want to fit more subplots to the same figure, add more `Axes` objects. Here is an example using some code from one of the official Matplotlib tutorials.

```
x = np.linspace(0, 2*np.pi, 100) # x values grid shared by both subplots

# create two subplots...one row two columns
fig, myAxes = plt.subplots(1, 2) # kind of like par(mfrow=c(1,2)) in R

# first subplot
myAxes[0].plot(x, x, label='linear')  # Plot some data on the axes.
## [<matplotlib.lines.Line2D object at 0x7ffcccfd2c18>]
myAxes[0].plot(x, x**2, label='quadratic')  # Plot more data on the axes...
## [<matplotlib.lines.Line2D object at 0x7ffccdfd9898>]
myAxes[0].plot(x, x**3, label='cubic')  # ... and some more.
## [<matplotlib.lines.Line2D object at 0x7ffccdfd9c18>]
myAxes[0].set_xlabel('x label')  # Add an x-label to the axes.
## Text(0.5, 0, 'x label')
myAxes[0].set_ylabel('y label')  # Add a y-label to the axes.
## Text(0, 0.5, 'y label')
myAxes[0].set_title("Simple Plot")  # Add a title to the axes.
## Text(0.5, 1.0, 'Simple Plot')
myAxes[0].legend()  # Add a legend.

# second subplot
## <matplotlib.legend.Legend object at 0x7ffccda55c18>
myAxes[1].plot(x,np.sin(x), label='sine wave')
## [<matplotlib.lines.Line2D object at 0x7ffccdf65240>]
myAxes[1].legend()
```

`r_and_python_book_files/figure-latex/unnamed-chunk-1`

import numpy as np my_inputs = np.linspace(start = 0, stop = 2*np.pi) outputs = np.sin(my_inputs)

import matplotlib.pyplot as plt plt.plot(my_inputs, outputs)

very good! https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html

3.6 Exercises

1. Write a Metropolis-Hastings algorithm.
2. Make a gif
3. What kind of situation would a left- or right-join be used?
4. Make a “phase plot”

Chapter 4

An Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a way of thinking about how to organize programs. This way of thinking focuses on objects. In the next chapter, we focus on organizing programs by functions, but for now we stick to objects. We already know about objects from the last chapter, so what's new here?

The difference is that we're creating our own *types* now. In the last chapter we learned about built-in types: floating point numbers, lists, arrays, functions, etc. Now we will discuss broadly how one can create his own types in both R and Python. These user-defined types can be used as cookie cutters. Once we have the cookie cutter, we can make as many cookies as we want!

TODO: image of cookie cutter

We will not go into this too deeply, but it is important to know how how code works so that we can use it more effectively. For instance, in Python, we frequently write code like `my_data_frame.doSomething()`. The material in this chapter will go a long way to describe how we can make our own types with custom behavior.

Here are a few abstract concepts that will help thinking about OOP. They are not mutually exclusive, and they aren't unique to OOP, but understanding these words will help you understand the purpose of OOP. Later on, when we start looking at code examples, I will alert you to when these concepts are coming into play.

- **Composition** refers to the idea when one type of object *contains* an object of another type. For example, a linear model object could hold onto estimated regression coefficients, residuals, etc.
- **Inheritance** takes place when an object can be considered to be of another type(s). For example, an analysis of variance linear regression model might be a special case of a general linear model.
- **Polymorphism** is the idea that the programmer can use the same code on objects of different types. For example, built-in functions in both R and Python can work on arguments of a wide variety of different types.
- **Encapsulation** is another word for complexity hiding. Do you have to understand every line of code in a package you're using? No, because a lot of details are purposefully hidden from you.
- **Modularity** is an idea related to encapsulation–it means splitting something into independent pieces. How you split code into different files, different functions, different classes–all of that has to do with modularity. It promotes encapsulation, and it allows you to think about only a few lines of code at a time.
- The **interface**, between you and the code you're using, describes *what* can happen, but not *how* it happens. In other words, it describes some functionality so that you can decide whether you want to use it, but there are not enough details for you to make it work yourself. For example, all you have to do to be able to estimate a complicated statistical model is to look up some documentation.[1] In other words, you only need to be familiar with the interface, not the implementation.

[1]Just because you can do this, doesn't mean you *should*, though!

- The **implementation** of some code you're using describes *how* it works in detail. If you are a package author, you can change your code's implementation "behind the scenes" and ideally, your end-users would never notice.

4.1 OOP In Python

4.1.1 Overview

In Python, classes are user-defined types. When you define your own class, you describe what kind of information it holds onto, and how it behaves.

To define your own type, use the `class` keyword. Objects created with a user-defined class are sometimes called **instances**. The behave according to the rules written in the class definition–they always have data and/or functions bundled together in the same way, but these instances do not all have the same data.

To be more clear, classes may have the following two things in their definition.

- **Attributes** are pieces of data "owned" by an instance created by the class.
- **(Instance) methods** are functions "owned" by an instance created by the class. They can use and/or modify data belonging to the class.

4.1.2 A First Example

Here's a simple example. Say we are interested in calculating, from numerical data $x_1, \ldots, x_n$, a sample mean:

$$\bar{x}_n = \frac{\sum_{i=1}^n x_i}{n}.$$

In Python, we can usually calculate this one number very easily using `np.average`. However, this function requires that we pass into it all of the data at once. What if we don't have all the data at any given time? In other words, suppose that the data arrive intermittently . We might consider taking advantage of a recursive formula for the sample means.

$$\bar{x}_n = \frac{(n-1)\bar{x}_{n-1} + x_n}{n}$$

How would we program this in Python? A first option: we might create a variable `my_running_ave`, and after every data point arrives, we could

```
my_running_ave = 1.0
my_running_ave
## 1.0
my_running_ave = ((2-1)*my_running_ave + 3.0)/2
my_running_ave
## 2.0
my_running_ave = ((3-1)*my_running_ave + 2.0)/3
my_running_ave
## 2.0
```

There are a few problems with this. Every time we add a data point, the formula slightly changes. Every time we update the average, we have to write a different line of code. This opens up the possibility for more bugs, and it makes your code less likely to be used by other people and more difficult to understand. And if we were trying to code up something more complicated than a running average? That would make matters even worse.

A second option: write a class that holds onto the running average, and that has

1. an `update` method that updates the running average every time a new data point is received, and
2. a `get_current_xbar` method that gets the most up-to-date information for us.

Using our code would look like this:

```
my_ave = RunningMean() # create running average object
my_ave.get_current_xbar() # no data yet!
my_ave.update(1.) # first data point
my_ave.get_current_xbar() # xbar_1
## 1.0
my_ave.update(3.)  # second data point
my_ave.get_current_xbar()  #xbar_2
## 2.0
my_ave.n   # my_ave.n instead of self.n
## 2
```

There is a Python convention that stipules class names should be written in `UpperCamelCase` (e.g. `RunningMean`).

That's much better! Notice the *encapsulation*. Looking at this code we don't need to think about the mathematical formula and the data being received. We only need to think about the latter. In other words, the *implementation* is separated from the *interface*. The interface in this case, is just the name of the class methods, and the arguments they expect. That's all we need to know about to use this code.

Classes (obviously) need to be defined before they are used, so here is the definition of our class.

```
class RunningMean:
  """Updates a running average"""
  def __init__(self):
    self.current_xbar = 0.0
    self.n = 0
  def update(self, new_x):
    self.n += 1
    self.current_xbar = (self.current_xbar*(self.n - 1) + new_x) / self.n
  def get_current_xbar(self):
    if self.n == 0:
      return None
    else:
      return self.current_xbar
```

Methods that look like `\_\_init\_\_`, or that possess names that begin and end with two underscores, are called **dunder (double underscore) methods**, **special methods** or **magic methods**. There are many that you can take advantage of! For more information see this.

Here are the details of the class definition:

1. Defining class methods looks exactly like defining functions! The primary difference is that the first argument must be `self`. If the definition of a method refers to `self`, then this allows the class instance to refer to its own (heretofore undefined) data attributes. Also, these method definitions are indented inside the definition of the class.

2. This class owns two data attributes. One to represent the number of data points seen up to now (`n`), and another to represent the current running average (`current_xbar`).

3. Referring to data members requires dot notation. `self.n` refers to the `n` belonging to any instance. This data attribute is free to vary between all the objects instantiated by this class.

4. The `__init__` method performs the setup operations that are performed every time any object is instantiated.

5. The `update` method provides the core functionality using the recursive formula displayed above.

6. `get_current_xbar` simply returns the current average. In the case that this function is called before any data has been seen, it returns `None`.

A few things you might find interesting:

i. Computationally, there is never any requirement that we must hold *all* of the data points in memory. Our data set could be infinitely large, and our class will hold onto only one floating point number, and one integer.

ii. This example is generalizable to other statistical methods. In a mathematical statistics course, you will learn about a large class of models having *sufficient statistics*. Most sufficient statistics have recursive formulas like the one above. Second, many algorithms in *time series analysis* have recursive formulas and are often needed to analyze large streams of data. They can all be wrapped into a class in a way that is similar to the above example.

4.1.3 Adding Inheritance

How can we use inheritance in statistical programming? A primary benefit of inheritance is code re-use, so one example of inheritance is writing a generic algorithm as a base class, and a specific algorithm as a class that inherits from the base class. For example, we could re-use the code in the `RunningMean` class in a variety of other classes.

Let's make some assumptions about a *parametric model* that is generating our data. Suppose I assume that the data points $x_1, \ldots, x_n$ are a "random sample"[2] from a normal distribution with mean μ and variance $\sigma^2 = 1$. μ is assumed to be unknown (this is, after all, and interval for μ), and σ^2 is assumed to be known, for simplicity.

A 95% confidence interval for the true unknown population mean μ is

$$\left[\bar{x} - 1.96\sqrt{\frac{\sigma^2}{n}}, \bar{x} + 1.96\sqrt{\frac{\sigma^2}{n}}\right].$$

The width of the interval shrinks as we get more data (as $n \to \infty$). We can write another class that, not only calculates the center of this interval, $\bar{x}$, but also returns the interval endpoints.

If we wrote another class from scratch, then we would need to rewrite a lot of the code that we already have in the definition of `RunningMean`. Instead, we'll use the idea of *inheritance*.

```python
import numpy as np
class RunningCI(RunningMean):
```

[2]Otherwise known as an independent and identically distributed sample

```
  """Updates a running average and gives you a known-variance confidence int
  def __init__(self, known_var):
    super().__init__()
    self.known_var = known_var
  def get_current_interval(self):
    if self.n == 0:
      return None
    else:
      half_width = 1.96 * np.sqrt(self.known_var / self.n)
      return np.array([self.current_xbar - half_width, self.current_xbar + hal
```

The parentheses in the first line of the class definition signal that this new class definition is inheriting from `RunningMean`. Inside the definition of this new class, when I refer to `self.current_xbar` Python knows what I'm referring to because it is defined in the base class. Last, I am using `super()` to access the base class's methods, such as `__init__`.

```
my_ci = RunningCI(1) # create running average object
my_ci.get_current_xbar() # no data yet!
my_ci.update(1.)
my_ci.get_current_interval()
## array([-0.96,  2.96])
my_ci.update(3.)
my_ci.get_current_interval()
## array([0.61407071, 3.38592929])
```

This example also demonstrates **polymorphism.** Polymorphism comes from the Greek for "many forms." "Forms" means "type" or "class" in this case. If the same code (usually a function or method) works on objects of different types, that's polymorphic. Here, the `update` method worked on an object of class `RunningCI`, as well as an object of `RunningMean`.

Why is this useful? Consider this example.

```
for datum in time_series:
  for thing in obj_list:
    thing.update(xt)
```

Inside the inner `for` loop, there is no need for include conditional logic that tests for what kind of type each `thing` is. We can iterate through time more succinctly.

```
for datum in time_series:
  for thing in obj_list:
    if isinstance(thing, class1):
      thing.updatec1(xt)
    if isinstance(thing, class2):
      thing.updatec2(xt)
    if isinstance(thing, class3):
      thing.updatec3(xt)
    if isinstance(thing, class4):
      thing.updatec4(xt)
    if isinstance(thing, class5):
      thing.updatec5(xt)
    if isinstance(thing, class6):
      thing.updatec6(xt)
```

If, in the future, you add a new class called `class7`, then you need to change this inner `for` loop, as well as provide new code for the class.

4.1.4 Adding in Composition

Composition also enables code re-use. Inheritance ensures an "is a" relationship between base and derived classes, and composition promotes a "has a" relationship. Sometimes it can be tricky to decide which technique to use, especially when it comes to statistical programming.

Regarding the example above, you might argue that a confidence interval isn't a particular type of a sample mean. Rather, it only *has a* sample mean. If you believe this, then you might opt for a composition based model instead. With composition, the derived class (the confidence interval class) will be decoupled from the base class (the sample mean class). This decoupling will have a few implications. In general, composition is more flexible, but can lead to longer, uglier code.

1. You will lose polymorphism.

2. Your code might become less re-usable.

    - You have to write any derive class methods you want because you don't inherit any from the base class. For example, you won't automatically get the `.update()` or the `.get_current_xbar()` method for free. This can be tedious if there are a lot of methods you want both classes to have that should work the same exact way for both classes. If there are, you would have to re-write a bunch of method definitions.
    - On the other hand, this could be good if you have methods that behave completely differently. Each method you write can have totally different behavior in the derived class, even if the method names are the same in both classes. For instance, `.update()` could mean something totally different in these two classes. Also, in the derive class, you can still call the base class's `.update()` method.

3. Many-to-one relationships are easier. It's generally easier to "own" many base class instances rather than inherit from many base classes at once. This is especially true if this is the only book on programming you plan on reading–I completely avoid the topic of multiple inheritance!

Sometimes it is very difficult to choose between using composition or using inheritance. However, this choice should be made very carefully. If you make the wrong one, and realize too late, *refactoring* your code might be very time consuming!

Here is an example implementation of a confidence interval using composition. Notice that this class "owns" a `RunningMean` instance called `self.mean`. This is contrast with *inheriting* from the `RunningMean` class.

```
class RunningCI2:
  """Updates a running average and gives you a known-variance confidence int
  def __init__(self, known_var):
    self.mean = RunningMean()
    self.known_var = known_var
  def update(self, new_x):
    self.mean.update(new_x)
  def get_current_interval(self):
    if self.n == 0:
      return None
    else:
      half_width = 1.96 * np.sqrt(self.known_var / self.n)
      return np.array([self.mean.get_current_xbar() - half_width, self.mean.get_curre
```

4.2 OOP In R

R, unlike Python, has many different kinds of classes. In R, there is not only one way to make a class. There are many! This list isn't exhaustive, but I will discuss

- S3 classes
- S4 classes
- Reference classes, and
- R6 classes.

If you like how Python does OOP, you will like reference classes and R6 classes, while S3 and S4 classes will feel strange to you.

It's best to learn about them chronologically, in my opinion. S3 classes came first, S4 classes sought to improve upon those. Reference classes rely on S4 classes, and R6 classes are an improved version of Reference classes.

TODO a picture of a wide variety of choices

4.2.1 S3 objects: The Big Picture

With S3 (and S4) objects, calling a method `print` will not look like this.

```
my_obj.print()
```

Instead, it will look like this:

```
print(my_obj)
```

The primary goal of S3 is *polymorphism*. We want functions like `print`, `summary` and `plot` to behave differently when objects of a different type are passed in to them. Printing a linear model should look a lot different than printing a data frame, right? So we can write code like the following, we only have to remember fewer functions as an end-user, and the "right" thing will always happen. If you're writing a package, it's also nice for your users that they're able to use the regular functions that they're familiar with. For instance, I allow users of my package `cPseudoMaRg` (Brown, 2021) to call `print` on objects of type `cpmResults`. In section 3.5.2, `ggplot2` instances, which are much more complicated than plain `numeric` `vector`s, are `+`ed together.

```
# print works on pretty much everything
print(myObj)
print(myObjOfADifferentClass)
print(aThirdClassObject)
```

This works because these "high-level" functions (like `print`), will look at its input and choose the most appropriate function to call, based on what kind of type the input has. `print` is the high-level function. When you run some of the above code, it might not be obvious which specific function `print` chooses for each input. You can't see that happening, yet.

Last, recall that this discussion only applies to S3 objects. Not all objects are S3 objects, though. To find out if an object `x` is an S3 object, use `is.object(x)`.

4.2.2 Using S3 objects

Using S3 objects is so easy that you probably don't even know that you're actually using them. You can just try to call functions on objects, look at the output, and if you're happy with it, you're good. However, if you've ever asked yourself: "why does `print` (for another function) do different things all the time?" then this section will be useful for you to read.

TODO: picture of looking into one door but going down a lot of avenues `print` is a **generic function** which is a function that looks at the type of its first argument, and then calls another, more specialized function depending on what type that argument is. Not all functions in `R` are generic, but some are. In addition to `print`, `summary` and `plot` are the most ubiquitous generic functions. Generic functions are an *interface*, because the user does not need to concern himself with the details going on behind the scenes.

In R, a **method** is a specialized function that gets chosen by the generic function for a particular type of input. The method is the *implementation*. When the generic function chooses a particular method, this is called **method dispatch**.

If you look at the definition of a generic function, let's take `plot` for instance, it has a single line that calls `UseMethod`.

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7ffcf78c6470>
## <environment: namespace:base>
```

`UseMethod` performs method dispatch. Which methods can be dispatched to? To see that, use the `methods` function.

```
methods(plot)
```

```
##   [1] plot,ANY-method
##   [2] plot,color-method
##   [3] plot.aareg*
##   [4] plot.acf*
##   [5] plot.agnes*
##   [6] plot.areg*
##   [7] plot.areg.boot*
##   [8] plot.aregImpute*
##   [9] plot.biVar*
## [10] plot.clusGap*
## [11] plot.cox.zph*
## [12] plot.curveRep*
## [13] plot.data.frame*
## [14] plot.decomposed.ts*
## [15] plot.default
## [16] plot.dendrogram*
## [17] plot.density*
## [18] plot.describe*
## [19] plot.diana*
## [20] plot.drawPlot*
## [21] plot.ecdf
## [22] plot.factor*
## [23] plot.formula*
## [24] plot.function
## [25] plot.gbayes*
## [26] plot.ggplot*
## [27] plot.gtable*
## [28] plot.hcl_palettes*
## [29] plot.hclust*
## [30] plot.histogram*
## [31] plot.HoltWinters*
## [32] plot.isoreg*
## [33] plot.lm*
## [34] plot.medpolish*
## [35] plot.mlm*
## [36] plot.mona*
## [37] plot.numpy.ndarray*
## [38] plot.partition*
## [39] plot.ppr*
## [40] plot.prcomp*
## [41] plot.princomp*
## [42] plot.profile.nls*
## [43] plot.Quantile2*
## [44] plot.R6*
## [45] plot.raster*
## [46] plot.rm.boot*
## [47] plot.rpart*
## [48] plot.shingle*
## [49] plot.silhouette*
## [50] plot.spec*
## [51] plot.spline*
## [52] plot.stepfun
## [53] plot.stl*
## [54] plot.summary.formula.response*
## [55] plot.summary.formula.reverse*
## [56] plot.summaryM*
## [57] plot.summaryP*
## [58] plot.summaryS*
## [59] plot.Surv*
## [60] plot.survfit*
## [61] plot.table*
## [62] plot.trans*
## [63] plot.transcan*
## [64] plot.trellis*
## [65] plot.ts
## [66] plot.tskernel*
## [67] plot.TukeyHSD*
## [68] plot.varclus*
## [69] plot.xyVector*
## see '?methods' for accessing help and source code
```

All of these S3 class methods share the same naming convention. Their name has the generic function's name as a prefix, then a dot (`.`), then the name of the class that they are specifically written to be used with.

R's dot-notation is nothing like Python's dot-notation! In R, functions do not *belong* to types/classes like they do in Python!

Method dispatch works by looking at the `class` attribute of an S3 object argument. An object in R may or may not have a set of **attributes**, which are a collection of name/value pairs that give a particular object extra functionality. Regular `vector`s in R don't have attributes (e.g. try running `attributes(1:3)`), but objects that are "embellished" versions of a `vector` might (e.g. run `attributes(factor(1:3))`).

`class` will return misleading results if it's called on an object that isn't an S3 object. Make sure to check with `is.object` first.

Also, these methods are not *encapsulated* inside a class definition like they are in Python, either. They just look like loose functions–the method definition for a particular class is not defined inside the class. These class methods can be defined just as ordinary functions, out on their own, in whatever file you think is appropriate to define functions in.

As an example, let's try to `plot` some specific objects.

```
aDF <- data.frame(matrix(rnorm(100), nrow = 10))
is.object(aDF) # is this s3?
## [1] TRUE
class(aDF)
## [1] "data.frame"
plot(aDF)
```

Because `aDF` has its `class` set to `data.frame`, this causes `plot` to try to find a `plot.data.frame` method. If this method was not found, R would attempt to find/use a `plot.default` method. If no default method existed, an error would be thrown.

As another example, we can play around with objects created with the `ecdf` function. This function computes an *empirical cumulative distribution function*, which takes a real number as an input, and outputs the proportion of observations that are less than or equal to that input[3]

```
myECDF <- ecdf(rnorm(100))
is.object(myECDF)
## [1] TRUE
class(myECDF)
## [1] "ecdf"     "stepfun"  "function"
plot(myECDF)
```

This is how *inheritance* works in R. The `ecdf` class inherits from the `stepfun` class, which in turn inherits from the `function` class. When you

[3]It's defined as $\hat{F}(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(X_i \le x)$.

call plot(myECDF), ultimately plot.ecdf is used on this object. However, if plot.ecdf did not exist, plot.stepfun would be tried. S3 inheritance in R is much simpler than Python's inheritance!

4.2.3 Creating S3 objects

Creating an S3 object is very easy, and is a nice way to spruce up some bundled up object that you're returning from a function, say. All you have to do is tag an object by changing its class attribute. Just assign a character vector to it!

Here is an example of creating an object of CoolClass.

```
myThing <- 1:3
attributes(myThing)
## NULL
class(myThing) <- "CoolClass"
attributes(myThing) # also try class(myThing)
## $class
## [1] "CoolClass"
```

myThing is now an instance of CoolClass, even though I never defined what a CoolClass was ahead of time! If you're used to Python, this should seem very strange. Compared to Python, this approach is very flexible, but also, kind of dangerous, because you can change the classes of existing objects. You shouldn't do that, but you could if you wanted to.

After you start creating your own S3 objects, you can write your own methods associated with these objects. That way, when a user of your code uses typical generic functions, such as summary, on your S3 object, you can control what interesting things will happen to the user of your package. Here's an example.

```
summary(myThing)
## [1] "No summary available!"
## [1] "Cool Classes are too cool for summaries!"
## [1] ":)"
```

The `summary` method I wrote for this class is the following.

```r
summary.CoolClass <- function(object,...){
  print("No summary available!")
  print("Cool Classes are too cool for summaries!")
  print(":)")
}
```

When writing this, I kept the signature the same as `summary`'s.

4.2.4 S4 objects: The Big Picture

S4 was developed after S3. If you look at your search path (type `search()`), you will see `package:methods`. That's where all the code you need to do S4 is.

Here are the similarities and differences between S3 and S4:

- they both use generic functions and methods work in the same way
- unlike in S3, S4 classes allow you to use multiple dispatch, which means the generic function can dispatch on multiple arguments, instead of just the first argument
- S4 class definitions are strict. They aren't just name tags like in S3.
- S4 inheritance feels more like Python's. You can think of a base class object living inside a child class object.
- S4 classes can have default data members via `prototypes`.

Much more information about S4 classes can be obtained by reading Chapter 15 in Hadley Wickham's book.

4.2.5 Using S4 objects

One CRAN package that uses S4 is the Matrix package. S4 objects are also extremely popular in packages hosted on Bioconductor.

Bioconductor is kind of like CRAN, but its software has a much more specific focus on bioinformatics. To download packages from Bioconductor, you can check out the installation instructions provided here.

```
library(Matrix)
M <- Matrix(10 + 1:28, 4, 7)
isS4(M)
## [1] TRUE
M
## 4 x 7 Matrix of class "dgeMatrix"
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]   11   15   19   23   27   31   35
## [2,]   12   16   20   24   28   32   36
## [3,]   13   17   21   25   29   33   37
## [4,]   14   18   22   26   30   34   38
M@Dim
## [1] 4 7
```

Inside an S4 object, data members are called **slots**, and they are accessed with the `@` operator (instead of the `$` operator). Objects can be tested if they are S4 with the function `isS4`. Otherwise, they look and feel just like S3 objects.

4.2.6 Creating S4 objects

Here are the key takeaways

- create a new S4 class with `setClass`
- create a new S4 object with `new`
- S4 classes have a fixed amount of slots, a name, and a fixed inheritance structure

Let's do an example that resembles the example we did in Python, where we defined a `RunningMean` class and a `RunningCI` class.

```
setClass("RunningMean",
         slots = list(n = "integer",
                      currentXbar = "numeric"))
setClass("RunningCI",
         slots = list(knownVar = "numeric"),
         contains = "RunningMean")
```

Here, unlike in S3 class's, we actually have to define a class with `setClass`. In the parlance of S4, `slots` are a class' data members, and `contains` signals that one class inherits from another (even though it kind of sounds like *composition*).

New objects can be created with the `new` function after this is accomplished.

```
new("RunningMean", n = 0L, currentXbar = 0)
new("RunningCI", n = 0L, currentXbar = 0, knownVar = 1.0)
```

Let's set one of those up. To achieve we want this `update` method to be generic, and it will work on objects of type "RunningMean" and "RunningCI".

Next we want to define an `update` generic function that will work on objects of both types. This is what gives us *polymorphism* . The generic `update` will call specialized methods for objects of class "RunningMean" and "RunningCI".

Recall that in the Python example, each class had its own update method. Here, we still have a specialized method for each class, but S4 methods don't have to be defined inside the class definition, as we can see below.

```
setGeneric("update", function(oldMean, newNum) {
  standardGeneric("update")
})
## [1] "update"
setMethod("update",
          c(oldMean = "RunningMean", newNum = "numeric"),
          function(oldMean, newNum) {
            oldN <- oldMean@n
            oldAve <- oldMean@currentXbar
            newAve <- (oldAve*oldN + newNum)/(oldN + 1)
            newN <- oldN + 1L
            return(new("RunningMean", n = newN, currentXbar = newAve))
          }
)
setMethod("update",
          c(oldMean = "RunningCI", newNum = "numeric"),
          function(oldMean, newNum) {
            oldN <- oldMean@n
            oldAve <- oldMean@currentXbar
            newAve <- (oldAve*oldN + newNum)/(oldN + 1)
            newN <- oldN + 1L
            return(new("RunningCI", n = newN, currentXbar = newAve, knownVar = oldMea
          }
)
```

Here's a demonstration of using these two classes that mirrors the example in subsection 4.1.3

```
myAve <- new("RunningMean", n = 0L, currentXbar = 0)
myAve <- update(myAve, 3)
myAve
## An object of class "RunningMean"
## Slot "n":
## [1] 1
##
## Slot "currentXbar":
## [1] 3
myAve <- update(myAve, 1)
myAve
## An object of class "RunningMean"
## Slot "n":
## [1] 2
##
## Slot "currentXbar":
## [1] 2

myCI <- new("RunningCI", n = 0L, currentXbar = 0, knownVar = 1.0)
myCI <- update(myCI, 3)
myCI
## An object of class "RunningCI"
## Slot "knownVar":
## [1] 1
##
```

```
## Slot "n":
## [1] 1
##
## Slot "currentXbar":
## [1] 3
myCI <- update(myCI, 1)
myCI
## An object of class "RunningCI"
## Slot "knownVar":
## [1] 1
##
## Slot "n":
## [1] 2
##
## Slot "currentXbar":
## [1] 2
```

This looks more *functional* (more information on functional programming is available in chapter 5) than the Python example because the `update` function does not change a *mutable* object with a side-effect. Instead, it takes the old object, changes it, returns the new object, and uses assignment to overwrite the object. The benefit of this aproach is that the assignment operator (`<-`) signals to us that something is changing. However, there is more data copying involved, so the program is presumably slower than it needs to be.

The big takeaway here is that S3 and S4 dont' feel like Python's encapsulated OOP. If we wanted to write stateful programs, we might consider using Reference Classes, or R6 classes.

4.2.7 Reference Classes: The Big Picture

Reference Classes. are built on top of S4 classes, and were released in late 2010. They feel very different from S3 and S4 classes, and they more closely resemble Python classes, because

1. their method definitions are *encapsulated* inside class definitions like in Python, and

2. the objects they construct are *mutable*.

So it will feel much more like Python's class system. Some might say using reference classes that will lead to code that is not very R-ish, but it can be useful for certain types of programs (e.g. long-running code, code that that performs many/high-dimensional/complicated simulations, or code that circumvents storing large data set in your computer's memory all at once).

4.2.8 Creating Reference Classes

Creating reference classes is done with the function `setRefClass`. I create a class called `RunningMean` that produces the same behavior as that in the previous example.

```
RunningMeanRC <- setRefClass("RunningMeanRC",
                           fields = list(current_xbar = "numeric",
                                         n = "integer"),
                           methods = list(
                             update = function(new_x){
                               n <<- n + 1L
                               current_xbar <<- (current_xbar*(n - 1) + new_x) / n
                             }))
```

This tells us a few things. First, data members are called *fields* now. Second, changing class variables is done with the `<<-`. We can use it just as before.

```
my_ave <- RunningMeanRC$new(current_xbar=0, n=0L)
my_ave
## Reference class object of class "RunningMeanRC"
## Field "current_xbar":
## [1] 0
## Field "n":
## [1] 0
my_ave$update(1.)
my_ave$current_xbar
```

```
## [1] 1
my_ave$n
## [1] 1
my_ave$update(3.)
my_ave$current_xbar
## [1] 2
my_ave$n
## [1] 2
```

Compare how similiar this code looks to the code in 4.1.2! Note the paucity of assignment operators, and plenty of side-effects.

4.2.9 Creating R6 Classes

I quickly implement the above example as an R6 class. A more detailed introduction to R6 classes is provided in the vignette from the package authors.

You'll notice the reappearance of the `self` keyword. R6 classes have a `self` keyword just like in Python. They are similar to reference classes, but there are a few differences:

1. they have better performance than reference classes, and
2. they don't make use of the `<<-` operator.

```
library(R6)

RunningMeanR6 <- R6Class("RunningMeanR6",
                  public = list(
                    current_xbar = NULL,
                    n = NULL,
                    initialize = function(current_xbar = NA, n = NA) {
                      self$current_xbar <- current_xbar
                      self$n <- n
                    },
                    update = function(new_x) {
                      self$n <- self$n + 1L
                      self$current_xbar <- (self$current_xbar*(self$n - 1) + new_x) /
                    }
                  )
)
my_r6_ave <- RunningMeanR6$new(current_xbar=0, n=0L)
my_r6_ave
## <RunningMeanR6>
##   Public:
##     clone: function (deep = FALSE)
##     current_xbar: 0
##     initialize: function (current_xbar = NA, n = NA)
##     n: 0
##     update: function (new_x)
my_r6_ave$update(1.)
my_r6_ave$current_xbar
## [1] 1
my_r6_ave$n
## [1] 1
my_r6_ave$update(3.)
my_r6_ave$current_xbar
## [1] 2
my_r6_ave$n
## [1] 2
```

TODO a more detailed

https://adv-r.hadley.nz/r6.html

https://adv-r.hadley.nz/r6.html#why-r6

4.3 Exercises

All answers to questions related to R should be written in a file named `data_types_exercises.R`. All answers to questions related to Python should be written in a file named `data_types_exercises.py`.

1. In Python, write a class that implements a running sample variance.‘

2. In Python, write a class that estimates a linear regression model.

Let

- Let $\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{10} \end{bmatrix}$ be a 10×1 vector of predictor variables.
- $\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{10} \end{bmatrix}$ be a 10×2 "design matrix",
- $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{10} \end{bmatrix}$ be a 10×1 vector of dependent observations, and
- $\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_{10} \end{bmatrix}$ be a 10×1 vector of mean 0 random errors. The assumed regression model can be written as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

and the estimate for the coefficient vector can be written as

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^\intercal \mathbf{X}\right)^{-1} \mathbf{X}^\intercal \mathbf{y}.$$

- Make the class called `LinearModel`, and use only the numpy package
- have the only attribute be a numpy array of estimated coefficients `coeffs`
- have one method called `fit` that takes a numpy arrays `X` (the design matrix), and `y`

- every time `fit` is called, reset the `coeffs` using the formula above

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

# don't hardcode variables!
num_rows = 10
num_predictors = 1
num_x_columns = num_predictors + 1

# generate fake data
true_coefficients = np.array([1,-3]).reshape((num_x_columns,1))
x_array = np.empty((num_rows, num_x_columns))
x_array[:,0] = 1
x_array[:,1] = np.random.normal(size=num_rows)
y_array = np.dot(x_array, true_coefficients)
y_array = y_array + np.random.normal(scale = .3, size = num_rows).reshape((num_rows,1

# plot fake data
plt.scatter(x_array[:,1], y_array)
```

`r_and_python_book_files/figure-latex/unnamed-chunk-201-1.pd`

Here is an example class definition, and then its use. The class holds onto

3. Come up with a list of Python classes that are written in third party libraries, and that you will find useful in the future!

4. Come up with a list of R classes that are written in third party libraries, and that you will find useful in the future!

5. Finish the R6 class example by adding a method `get_current_xbar` that takes no arguments, and returns `NULL` if `n` is equal to 0, and n otherwise. This class should behave exactly the same as the Python class that goes by the same name.

6. Come up with some ideas for (base class, derived class, object) triples.

7. Come up with some examples of when composition would be better than inhertiance, and vice versa.

Chapter 5

Functional Programming

Functional Programming (FP) is another way of thinking about how to organize programs. We talked about OOP in the last chapter (chapter 4), so how do OOP and FP differ? To put it simply, FP focuses on functions instead of objects. Because we are talking a lot about functions in this chapter, we will assume you have read and understood section 2.5.

Neither R nor Python is a purely functional language. For us, FP is a style that we can choose to let guide us, or that we can disregard. You can choose to employ a more functional style, or you can choose to use a more object-oriented style, or neither. Some people tend to prefer one style to other styles, and others prefer to decide which to use depending on the task at hand.

More specifically, a functional programming style takes advantage of **first-class functions** and favors functions that are **pure**.

1. **First-class functions** are functions that
    - can be passed as arguments to other functions,
    - can be returned from other functions, and
    - can be assigned to variables or stored in data structures.
2. **Pure functions**
    - return the same output if they are given the same input, and
    - do not produce **side-effects**.

Side-effects are changes made to non-temporary variables, to the "state" of the program.

We discussed (1) in the beginning of chapter 2.5. If you have not used any other programming languages before, you might even take (1) for granted. However, if you have written C code before, you might remember how difficult it is to use functions as inputs to other functions!

There is more to say about point (2). This means you should keep your functions as *modular* as possible, unless you want your overall program to be much more difficult to understand. FP stipulates that

- **ideally functions will not refer to non-local variables;**
- **ideally functions will not (refer to and) modify non-local variables; and**
- **ideally functions will not modify their arguments.**

Unfortunately, violating the first of these three criteria is very easy to do in both of our languages. Recall our conversation about *dynamic lookup* in subsection 2.5.8. Both R and Python use dynamic lookup, which means you can't reliably control *when* functions look for variables. Typos in variable names easily go undiscovered, and modified global variables can potentially wreak havoc on your overall program.

Fortunately it is difficult to modify global variables inside functions in both R and Python. This was also discussed in subsection 2.5.8. In Python, you need to make use of the `global` keyword (mentioned in section 2.5.7.2), and in R, you need to use the rare super assignment operator (it looks like `<<-`, and it was mentioned in 2.5.7.1). Because these two symbols are so rare, they can serve as signals to viewers of your code about when and where (in which functions) global variables are being modified.

Last, violating the third criterion is easy in Python and difficult in R. This was dicussed earlier in 2.5.7. Python can mutate/change arguments that have a mutable type because it has *pass-by-assignment* semantics (mentioned in section 2.5.7.2, and R generally can't modify its arguments at all because it has *pass-by-value* semantics 2.5.7.1.

This chapter avoids the philosophical discussion of FP. Instead, it takes the applied approach, and provides instructions on how to use FP in your own programs. I try to give examples of *how* you can use FP, and *when* these tools are especially suitable.

One of the biggest tip-offs that you should be using functional programming is if you need to evaluate a single function many times, or in many different ways. This happens quite frequently in statistical computing. Instead of copy/pasting similar-looking lines of code, you might consider *higher-order* functions that take your function as an input, and intelligently call it in all the many ways you want it to. A third option you might also consider is to use a loop (c.f. 3.3.2). However, that approach is not very functional, and so it will not be heavily-discussed in this section.

Another tip-off that you need FP is if you need many different functions that are all "related" to one another. Should you define each function separately, using excessive copy/paste-ing? Or should you write a function that can elegantly generate any function you need?

Not repeating yourself and re-using code is a primary motivation, but it is not the only one. Another motivation for **functional programming** is clearly explained in Advanced R[1]:

> A functional style tends to create functions that can easily be analysed in isolation (i.e. using only local information), and hence is often much easier to automatically optimise or parallelise.

All of these sound like a good things to have in our code, so let's get started with some examples!

5.1 Functions as Function Inputs in R

Many of the most commonly-used functionals in R have names that end in "apply". The ones I discuss are `sapply`, `vapply`, `lapply`, `apply`, `tapply` and `mapply`. Each of these takes a function as one of its arguments. Recall that this is made possible by the fact that R has first-class functions.

[1]Even though this book only discusses *one* of our languages of interest, this quote applies to both langauges.

5.1.1 sapply and vapply

Suppose we have a `data.frame` that has 10 rows and 100 columns. What if we want to take the mean of each column?

An amateurish way to do this would be something like the following.

```
myFirstMean <- mean(myDF[,1])
mySecondMean <- mean(myDF[,2])
myThirdMean <- mean(myDF[,3])
# ....
# so on and so forth
# ....
myThirdMean <- mean(myDF[,100])
```

You will need one line of code for each column in the data frame. For data frames with a lot of columns, this becomes quite tedious. You should also ask yourself what happens to you and your collaborators when the data frame changes even slightly, or if you want to apply a different function to its columns. Third, the results are not stored in a single container. You are making it difficult on yourself if you want to use these variables in subsequent pieces of code.

"Don't repeat yourself" (DRY) is an idea that's been around for a while and is widely accepted [@hunt2000pragmatic]. DRY is the opposite of WET.

Instead, prefer the use of `sapply` in this situation. The "s" in `sapply` stands for "simplified." In this bit of code `mean` is called on each column of th data frame. `sapply` applies the function over columns, instead of rows, because data frames are internally a `list` of columns.

```
myMeans <- sapply(myDF, mean)
head(myMeans)
##       X1       X2       X3       X4       X5       X6 
##  0.08435 -0.09955  0.41944 -0.11255 -0.06780 -0.21552
```

Each call to `mean` returns a `double vector` of length 1. This is necessary if you want to collect all the results into a `vector`–remember, all elements of a `vector` have to have the same type. To get the same behavior, you might also consider using `vapply(myDF, mean, numeric(1))`.

In the above case, "simplify" referred to how one-hundred length-1 vectors were simplified into one length-100 `vector`. However, "simplified" does not necessarily imply that all elements will be stored in a `vector`. Consider the `summary` function, which returns a `double` `vector` of length 6. In this case, one-hundred length-6 vectors were simplified into one 6×100 matrix.

```
mySummaries <- sapply(myDF, summary)
is.matrix(mySummaries)
## [1] TRUE
dim(mySummaries)
## [1]   6 100
```

Another function that is worth mentioning is `replicate`–it is a wrapper for `sapply`. Consider a situation where you want to call a function many times with the same inputs. You might try something like `sapply(1:100, function(elem) {  return(myFunc(someInput)) } )`. Another, more readable, way to do this is `replicate(100, myFunc(someInput))`.

5.1.2 `lapply`

For functions that do not return amenable types that fit into a `vector`, `matrix` or `array`, they might need to be stored in `list`. In this situation, you would need `lapply`. The "l" in `lapply` stands for "list". `lapply` always returns a `list` of the same length as the `input`.

```
regress <- function(y){ lm(y ~ 1) }
myRegs <- lapply(myDF, regress)
length(myRegs)
## [1] 100
class(myRegs[[1]])
## [1] "lm"
summary(myRegs[[12]])
##
## Call:
```

```
## lm(formula = y ~ 1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.1146 -0.5107 -0.0697  0.6059  1.2518
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.242      0.245    0.99     0.35
##
## Residual standard error: 0.774 on 9 degrees of freedom
```

5.1.3 `apply`

I use `sapply` and `lapply` the most, personally. The next most common function I use is `apply`. I use it to apply functions to *rows* instead of columns. However, it can also apply functions over columns, just as the other functions we discussed can.[2]

```
dim(myDF)
## [1]  10 100
apply(myDF, 1, mean)
##  [1]  0.0224854  0.0882708  0.0456250  0.1325681
##  [5] -0.0457316  0.2293268  0.0831142 -0.0007563
##  [9] -0.0136159  0.0632788
```

Another example where it can be useful to apply a function to rows is **predicate functions.** A predicate function is just a fancy name for a function that returns a Boolean. I use them to filter out rows of a `data.frame`. Without a predicate function, filtering rows might look something like this.

[2]`apply` is everyone's favorite whipping boy whenever it comes to comparing `apply` against the other `*apply` functions. This is because it is generally a little slower–it is written in R and doesn't call out to compiled C code. However, in my humble opinion, it doesn't matter all that much because the fractions of a second saved don't always add up in practice.

```
albRealEstate <- read.csv("data/albemarle_real_estate.csv")
subDF <- albRealEstate[(albRealEstate$YearBuilt == 2006 & albRealEstate$Condition ==
head(subDF)
##   YearBuilt YearRemodeled Condition NumStories FinSqFt
## 1      2006             0   Average       1.00    1922
## 2      2003             0   Average       1.00    1848
## 4      1998             0      Good       1.00    1244
## 5      1886             0   Average       1.86    1861
## 6      1910             0      Fair       1.53    1108
## 8      1975          1982   Average       1.00    1520
##   Bedroom FullBath HalfBath TotalRooms LotSize
## 1       3        3        0         10   5.000
## 2       3        2        0          7  61.189
## 4       1        1        0          3  50.648
## 5       4        1        0          6   3.880
## 6       3        1        0          6   8.838
## 8       3        1        0          4   2.030
##   TotalValue   City
## 1     409900 CROZET
## 2     523100 CROZET
## 4     620700 CROZET
## 5     162500 CROZET
## 6     167200 CROZET
## 8     151900 CROZET
```

Complicated filtering criteria can become quite wide, so I prefer to break the above code into three steps.

- Step 1: write a predicate function that returns TRUE or FALSE;
- Step 2: construct a `logical vector` by `apply`ing the predicate over rows;
- Step 3: plug the `logical vector` into the `[` operator to remove the rows.

```
pred <- function(row){
  (row['YearBuilt'] == 2006 & row['Condition'] == "Average") | row['City'] == "CROZET
}
```

```
whichRows <- apply(albRealEstate, 1, pred)
subDF <- albRealEstate[whichRows,]
head(subDF)
##   YearBuilt YearRemodeled Condition NumStories FinSqFt
## 1      2006             0   Average       1.00    1922
## 2      2003             0   Average       1.00    1848
## 4      1998             0      Good       1.00    1244
## 5      1886             0   Average       1.86    1861
## 6      1910             0      Fair       1.53    1108
## 8      1975          1982   Average       1.00    1520
##   Bedroom FullBath HalfBath TotalRooms LotSize
## 1       3        3        0         10   5.000
## 2       3        2        0          7  61.189
## 4       1        1        0          3  50.648
## 5       4        1        0          6   3.880
## 6       3        1        0          6   8.838
## 8       3        1        0          4   2.030
##   TotalValue   City
## 1     409900 CROZET
## 2     523100 CROZET
## 4     620700 CROZET
## 5     162500 CROZET
## 6     167200 CROZET
## 8     151900 CROZET
```

5.1.4 `tapply`

`tapply` can be very handy when you need it. First, we've alluded to the definition before in subsection 2.7.1, but a **ragged array** is a collection of arrays that all have potentially different lengths. I don't typically construct such an object and then pass it to `tapply`. Rather, I let `tapply` construct the ragged array for me. The first argument it expects is "typically vector-like", while the second tells us how to break that `vector` into chunks. The third argument is a function that gets applied to each `vector` chunk.

If I wanted the average home price for each city, I could use something like this.

```
head(albRealEstate)
##   YearBuilt YearRemodeled Condition NumStories FinSqFt
## 1      2006             0   Average       1.00    1922
## 2      2003             0   Average       1.00    1848
## 3      1972             0   Average       1.00    1248
## 4      1998             0      Good       1.00    1244
## 5      1886             0   Average       1.86    1861
## 6      1910             0      Fair       1.53    1108
##   Bedroom FullBath HalfBath TotalRooms LotSize
## 1       3        3        0         10   5.000
## 2       3        2        0          7  61.189
## 3       2        1        0          4   1.760
## 4       1        1        0          3  50.648
## 5       4        1        0          6   3.880
## 6       3        1        0          6   8.838
##   TotalValue        City
## 1     409900      CROZET
## 2     523100      CROZET
## 3     180900 EARLYSVILLE
## 4     620700      CROZET
## 5     162500      CROZET
## 6     167200      CROZET
unique(albRealEstate$City)
## [1] "CROZET"          "EARLYSVILLE"
## [3] "CHARLOTTESVILLE" "SCOTTSVILLE"
## [5] "NORTH GARDEN"    "KESWICK"
tapply(albRealEstate$TotalValue, list(albRealEstate$City), mean)
## CHARLOTTESVILLE          CROZET     EARLYSVILLE
##          381933          380426          439141
##         KESWICK    NORTH GARDEN     SCOTTSVILLE
##          540533          366598          268407
```

You might be wondering why we put `albRealEstate$City` into a `list`. That seems kind of unnecessary. This is because `tapply` can be used with multiple `factors`–this will break down the `vector` input into a finer partition. The second argument must be one object, though, so all of these `factors` must be collected into a `list`. The following code produces a "pivot table."

```
tapply(albRealEstate$TotalValue, list(albRealEstate$City, albRealEstate$Condit
##                  Average Excellent   Fair   Good   Poor
## CHARLOTTESVILLE  337913    491702 229336 444326 202420
## CROZET           342133    552082 198009 390658 203806
## EARLYSVILLE      365991    652387 230646 470555 372443
## KESWICK          392999    871720 172791 672511 100338
## NORTH GARDEN     241188    966529 131997 440503 151188
## SCOTTSVILLE      214047    502274 157438 374600  95378
##                  Substandard
## CHARLOTTESVILLE       457500
## CROZET                 53450
## EARLYSVILLE           160400
## KESWICK                   NA
## NORTH GARDEN              NA
## SCOTTSVILLE               NA
```

For functions that return higher-dimensional output, you will have to use something like `by` or `aggregate` in place of `tapply`.

5.1.5 mapply

The documentation of `mapply` states `mapply` is a multivariate version of `sapply`. `sapply` worked with univariate functions; the function was called multiple times, but each time with a single argument. If you have a function that takes multiple arguments, and you want those arguments to change each time the function is called, then you might be able to use `mapply`.

Here is a short example. Regarding the `n=` argument of `rnorm`, the documentation explains, "[i]f `length(n) > 1`, the length is taken to be the number required." This would be a problem if we want to sample a.) three times from a mean 0 normal, b.) twice from a mean 100 normal, and c.) once from a mean -100 normal distribution.

```
rnorm(n = c(3,2,1), mean = c(0,100,-100), sd = c(.01, .01, .01))
```

```
## [1] -1.441e-03  9.999e+01 -1.000e+02
```

```
mapply(rnorm, n = c(3,2,1), mean = c(0,100,-100), sd = c(.01, .01, .01))
```

```
## [[1]]
## [1]  0.007884  0.005248 -0.023179
##
## [[2]]
## [1] 100 100
##
## [[3]]
## [1] -99.99
```

5.1.6 Reduce and do.call

In section 3.4 we talked about several different ways of "combining" data sets. We discussed stacking data sets on top of one another with `rbind` (c.f. subsection 3.4.2), stacking them side-by-side with `cbind` (also in 3.4.2), and intelligently joining them together with `merge` (c.f. 3.4.3). Consider the task of combining *many* data sets. How do we write DRY code and abide by the DRY principle?

We can use either `Reduce` or `do.call` as a higher-order function. Just like the aforementioned `*apply` functions, they take in either `cbind`, `rbind`, or `merge` as a function input. Which one do we pick, though? The answer to that question deals with how many arguments our lower-order functions take.

Take a look at the documentation to `rbind`. Its first argument is `...`, which is the dot-dot-dot symbol. This means `rbind` can take a varying number of `data.frame`s to stack on top of each other. In other words, `rbind` is *variadic*.

On the other hand, take a look at the documentation of `merge`. It only takes two `data.frame`s at a time[3].

If we want to combine many data sets, `merge` This is the difference between `Reduce` and `do.call`.

[3] Although, it is still variadic. The difference is that the `dot-dot-dot` symbol does not refer to a varying number of `data.frame`s. . . just a varying number of other things we don't care about at the present moment.

`do.call` calls a function once on many arguments, so its function must be able to handle many arguments. On the other hand, `Reduce` calls a binary function many times on pairs of arguments. `Reduce`'s function argument gets called on the first two elements, then on the first output and the third element, then on the second output and fourth element, and so on.

TODO diagram

Here is an initial example that makes use of four data sets `d1.csv`, `d2.csv`, `d3.csv`, and `d4.csv`. To start, ask yourself how we would read all of these in. There is a temptation to copy and paste `read.csv` calls, but that would violate the DRY principle. Instead, let's use `lapply` an anonymous function that constructs a file path string, and then uses it to read in the data set the string refers to.

```
numDataSets <- 4
dataSets <- paste0("d",1:numDataSets)
dfs <- lapply(dataSets,
              function(name) read.csv(paste0("data/", name, ".csv")))
head(dfs[[3]])
##   id obs3
## 1  a    7
## 2  b    8
## 3  c    9
```

Notice how the above code would only need to be changed by one character if we wanted to increase the number of data sets being read in!

Next, `cbind`ing them all together can be done as follows. `do.call` will call the function only once. `cbind` takes many arguments at once, so this works.

```
do.call(cbind, dfs) # DRY! :)
##   id obs1 id obs2 id obs3 id obs4
## 1  a    1  b    5  a    7  a   10
## 2  b    2  a    4  b    8  b   11
## 3  c    3  c    6  c    9  c   12
# cbind(df1,df2,df3,df4) # WET! :(
```

This code is even better than the above code in that if `dfs` becomes longer, or changes at all, *nothing* will need to be changed.

What if we wanted to `merge` all these data sets together? After all, the `id` column appears to be repeating itself, and some data from `d2` isn't lining up.

```
Reduce(merge, dfs)
##   id obs1 obs2 obs3 obs4
## 1  a    1    4    7   10
## 2  b    2    5    8   11
## 3  c    3    6    9   12
```

Again, this is very DRY code. Nothing would need to be changed if `dfs` grew. Furthermore, trying to `do.call` the `merge` function wouldn't work because it can only take two data sets at a time.

5.2 Another Example in R

Consider another common example: plotting scalar-valued multivariate functions. Let's try to plot a bivariate Gaussian distribution.

$$f(x, y) = \frac{1}{2\pi} \exp\left[-\frac{x^2 + y^2}{2}\right]$$

The random elements x and y, in this particular case, are uncorrelated, each have unit variance, and zero mean. This density is a surface in 3-d dimensional space. To visualize this, we would need to

1. generate a "grid" of points in $\mathbb{R}^2$,
2. evaluate our function on each point, and then
3. call some plotting function that takes this all and makes a pretty picture.

There are a couple ways you could write this function. One way might take two arguments, and another might take one argument. If we are to use `mapply`, we need the function to take two arguments.

```
fTwoArgs <- function(x,y){
  exp(-.5*(x^2 + y^2)) / 2 / pi
}
```

We can construct every possible point on a grid with the `expand.grid` function.

```
xGrid <- seq(-3,3,.1)
yGrid <- seq(-3,3,.1)
grid <- expand.grid(xGrid, yGrid)
head(grid)
##   Var1 Var2
## 1 -3.0   -3
## 2 -2.9   -3
## 3 -2.8   -3
## 4 -2.7   -3
## 5 -2.6   -3
## 6 -2.5   -3
```

`mapply` would take `fTwoArgs`, and effectively call it on every row pair. The pairs do not need to be organized in a `data.frame`, though.

```
funcOut1 <- mapply(fTwoArgs, grid[,1], grid[,2])
head(funcOut1)
## [1] 1.964e-05 2.638e-05 3.508e-05 4.618e-05 6.020e-05
## [6] 7.768e-05
rectangularOutput <- matrix(funcOut1, ncol = length(xGrid))
persp(xGrid, yGrid, rectangularOutput,
      zlab = "f(x,y)", xlab = "x", ylab = "y")
```

r_and_python_book_files/figure-latex/unnamed-chunk-2

If you prefer using `apply`, that is also possible, but you would need to rewrite the function to take one (length-two) argument.

```
fOneArg <- function(vec){
  exp(-sum(vec^2)/2)/2/pi
}
funcOut2 <- apply(grid, 1, fOneArg)
moreRectOut <- matrix(funcOut2, ncol = length(xGrid))
contour(xGrid, yGrid, moreRectOut, xlab = "x", ylab = "y")
```

r_and_python_book_files/figure-latex/unnamed-chunk-222-1.pd

5.3 Functions as Function Inputs in Base Python

I discuss two functions from base Python that take functions as input. Neither return a `list` or a `np.array`, but they do return different kinds of **iterables**, which are "objects capable of returning their members one at a time," according to the Python documentation. `map`, the function, will return objects of type `map`. `filter`, the function, will return objects of type `filter`. Often times we will just convert these to the container we are more familiar with.

5.3.1 `map`

`map` can call a function repeatedly using elements of a container as inputs. Here is an example of calculating outputs of a *spline* function, which can be useful for coming up with predictors in regression models. This particular spline function is $f(x) = (x - k)1(x \geq k)$, where k is some chosen "knot point."

```
import numpy as np
my_inputs = np.linspace(start = 0, stop = 2*np.pi)
```

```
def spline(x):
    knot = 3.0
    if x >= knot:
        return x-knot
    else:
        return 0.0
output = list(map(spline, my_inputs))
```

We can visualize the mathematical function by plotting its outputs against its inputs. More information on visualization was given in subsection 3.5.

`map` can also be used like `mapply`. In other words, you can apply it to two containers,

```
import numpy as np
x = np.linspace(start = -1., stop = 1.0)
y = np.linspace(start = -1., stop = 1.0)
def f(x):
    np.log(x**2 + y**2)
output = list(map(spline, my_inputs))
```

5.3.2 `filter`

`filter` helps remove unwanted elements from a container. It returns an iterable of type `filter`, which we can iterate over or convert to a more familiar type of container. In this example, I iterate over it without converting it.

```
raw_data = np.arange(0,1.5,.01)
for elem in filter(lambda x : x**2 > 2, raw_data):
    print(elem)
## 1.42
## 1.43
## 1.44
## 1.45
## 1.46
## 1.47
## 1.48
## 1.49
```

5.4 Functions as Function Inputs in Numpy

Numpy provides a number of functions that facilitate working with `np.ndarray`s in a functional style. For example, `np.apply_along_axis` is similar to R's `apply`. `apply` had a `MARGIN` argument (`1` sums rows, `2` sums columns), whereas this function has a `axis=` argument (`0` sums columns, `1` sums rows).

```
import numpy as np
my_array = np.arange(6).reshape((2,3))
my_array
## array([[0, 1, 2],
##        [3, 4, 5]])
np.apply_along_axis(sum, 0, my_array) # summing columns
## array([3, 5, 7])
np.apply_along_axis(sum, 1, my_array) # summing rows
## array([ 3, 12])
```

```
my_array = np.random.normal(size=(10,1000))
np.apply_along_axis(sum, 0, my_array).shape
## (1000,)
np.apply_along_axis(sum, 1, my_array).shape
## (10,)
```

5.5 Functional Methods in Pandas

Pandas `DataFrame`s have an `.apply` method that is very similar to `apply` in R,[4] but again, just like the above function, you have to think about an `axis=` argument instead of a `MARGIN=` argument.

```
import pandas as pd
alb_real_est = pd.read_csv("data/albemarle_real_estate.csv")
alb_real_est.shape
## (27943, 12)
alb_real_est.apply(len, axis=0) # length of columns
## YearBuilt        27943
## YearRemodeled    27943
## Condition        27943
## NumStories       27943
## FinSqFt          27943
## Bedroom          27943
## FullBath         27943
## HalfBath         27943
## TotalRooms       27943
## LotSize          27943
## TotalValue       27943
## City             27943
## dtype: int64
type(alb_real_est.apply(len, axis=1)) # length of rows
## <class 'pandas.core.series.Series'>
```

Another thing to keep in mind is that `DataFrame`s, unlike `ndarray`s, don't have to have the same type for all elements. If you have mixed column types, then summing rows, for instance, might not make sense. This just requires subsetting columns before `.apply`ing a function to rows. Here is an example of computing each property's "score".

[4] You should know that a lot of special-case functions that you typically apply to rows or columns come built-in as `DataFrame` methods. For instance, `.mean` would allow you to do something like `my_df.mean()`.

```
import pandas as pd
# alb_real_est.apply(sum, axis=1) # can't add letters to numbers!
def get_prop_score(row):
  return 2*row[0] + 3*row[1]
alb_real_est['Score'] = alb_real_est[['FinSqFt','LotSize']].apply(get_prop_score, 1)
alb_real_est[['FinSqFt','LotSize','Score']].head(2)
##    FinSqFt  LotSize     Score
## 0     1922    5.000  3859.000
## 1     1848   61.189  3879.567
```

`.apply` also works with more than one function at a time.

```
alb_real_est[['FinSqFt','LotSize']].apply([sum, len])
##        FinSqFt     LotSize
## sum   55559801  97856.8384
## len      27943  27943.0000
```

If you do not want to waste two lines defining a function with `def`, you can use an anonymous (unnamed) **lambda function**. Be careful, though–if your function is complex enough, then your lines will get quite wide. For instance, this example is pushing it.

```
alb_real_est[['FinSqFt','LotSize']].apply(lambda row : sum(row*[2,3]), 1)[:4]
## 0    3859.000
## 1    3879.567
## 2    2501.280
## 3    2639.944
## dtype: float64
```

If you want to apply a (scalar-valued) function that takes only individual elements, you should try to use a unary function (recall that this was discussed in TODO). If no such unary function exists, you can apply it with `.applymap`.

```
alb_real_est[['FinSqFt','LotSize']].applymap(lambda e : e + 1).head(2)
##    FinSqFt  LotSize
## 0     1923    6.000
## 1     1849   62.189
```

Last, we have a `.groupby` method, which can be used to mirror the behavior of R's `tapply`, `aggregate` or `by`. It can take the `DataFrame` it belongs to, and group its rows into multiple sub-`DataFrame`s. The collection of sub-`DataFrames` has a lot of the same methods that an individual `DataFrame` has (e.g. the subsetting operators, and the `.apply` method), which can all be used in a second step of calculating things on each sub-`DataFrame`.

```
type(alb_real_est.groupby(['City']))
## pandas.core.groupby.generic.DataFrameGroupBy
type(alb_real_est.groupby(['City'])['TotalValue'])
## pandas.core.groupby.generic.SeriesGroupBy
```

Here is an example that models some pretty typical functionality. It shows two ways to get the average home price by city. The first line groups the rows by which `City` they are in, extracts the `TotalValue` column in each sub-`DataFrame`, and then `.apply`s the `np.average` function on the sole column found in each sub-`DataFrame`. The second applys a lambda function to each sub-`DataFrame` directly.

```
alb_real_est.groupby(['City'])['TotalValue'].apply(np.average)
## City
## CHARLOTTESVILLE    381932.962760
## CROZET             380425.678927
## EARLYSVILLE        439140.987124
## KESWICK            540532.605905
## NORTH GARDEN       366597.750865
## SCOTTSVILLE        268407.384615
## Name: TotalValue, dtype: float64
alb_real_est.groupby(['City']).apply(lambda df : np.average(df['TotalValue']))
## City
## CHARLOTTESVILLE    381932.962760
## CROZET             380425.678927
## EARLYSVILLE        439140.987124
## KESWICK            540532.605905
## NORTH GARDEN       366597.750865
## SCOTTSVILLE        268407.384615
## dtype: float64
```

More tips on this programming pattern can be found here.

5.6 Functions as Function Inputs (miscellany)

functools.reduce

operator module

5.7 Functions as Function Outputs in R

Functions that create and return other functions are sometimes called **function factories.** Functions are first-class objects in R, so it's easy to return them. What's more interesting is that supposedly temporary objects inside the outer function can be accessed during the call of the inner function after it's returned.

Here is a first quick example.

```
funcFactory <- function(greetingMessage){
  function(name){
    paste(greetingMessage, name)
  }
}
greetWithHello <- funcFactory("Hello")
greetWithHello("Taylor")
## [1] "Hello Taylor"
```

```
greetWithHello("Charlie")
## [1] "Hello Charlie"
```

The `greetingMessage` argument that is passed in, `"Hello"`, isn't temporary anymore.

Here is an example that implements a variance reduction technique called **common random numbers**.

Suppose $X \sim \text{Normal}(\mu, \sigma^2)$, and we are interested in approximating an expectation of a function of this random variable. Suppose that we don't know that

$$\mathbb{E}[\sin(X)] = \sin(\mu)\exp\left(-\frac{\sigma^2}{2}\right)$$

for any particular choice of μ and σ^2, and instead, we choose to use the Monte Carlo method:

$$\hat{\mathbb{E}}[\sin(X)] = \frac{1}{n}\sum_{i=1}^{n}\sin(X^i)$$

where $X^1, \ldots, X^n \overset{\text{iid}}{\sim} \text{Normal}(\mu, \sigma^2)$ is a large collection of draws from the appropriate normal distribution. In real life, the theoretical expectation might not be tractable (either because the random variable has a complicated distribution, or maybe because the functional is very complicated) and Monte Carlo, or some other approximation algorithm, might be our only hope!

Here are two functions that calculate the above quantities for $n = 100$. `actualExpectSin` is a function that computes the theoretical expectation for any particular parameter pair. `monteCarloSin` is a function that implements the Monte Carlo approximate expectation.

```
n <- 1000 # don't hardcode variables that aren't passed as arguments!
actualExpectSin <- function(params){
  stopifnot(params[2] > 0) # second parameter is sigma
  sin(params[1])*exp(-.5*(params[2]^2))
}
monteCarloSin <- function(params){
  stopifnot(params[2] > 0)
  mean(sin(rnorm(n = n, mean = params[1], sd = params[2])))
}
# monteCarloSin(c(10,1))
```

One-off approximations aren't as interesting as visualizing many expectations for many parameter inputs. On the left, we have the true expectation function plotted with a contour plot. On the right,

```
muGrid <- seq(-10,10, length.out = 100)
sigmaGrid <- seq(.001, 5, length.out = 100)
muSigmaGrid <- expand.grid(muGrid, sigmaGrid)
actuals <-  matrix(apply(muSigmaGrid, 1, actualExpectSin), ncol = length(muGrid))
mcApprox <- matrix(apply(muSigmaGrid, 1, monteCarloSin), ncol = length(muGrid))

par(mfrow=c(1,2))
contour(muGrid, sigmaGrid, actuals, xlab = "mu", ylab = "sigma", main = "actual expec
contour(muGrid, sigmaGrid, mcApprox, xlab = "mu", ylab = "sigma", main = "mc without
```

r_and_python_book_files/figure-latex/unnamed-chunk-238-1.pd

```
par(mfrow=c(1,1))
```

If we wanted to use common random numbers, we could generate $Z^1, \ldots, Z^n \overset{\text{iid}}{\sim} \text{Normal}(0,1)$, and use the fact that

$$X^i = \mu + \sigma Z^i$$

This leads to the Monte Carlo estimate

$$\tilde{\mathbb{E}}[\sin(X)] = \frac{1}{n}\sum_{i=1}^{n} \sin(\mu + \sigma Z^i)$$

Here is one function that naively implements Monte Carlo with common random numbers. We generate the collection of standard normal random variates once, globally. Each time you call `monteCarloSinCRNv1(c(10,1))`, you get the same answer.

```r
commonZs <- rnorm(n=n)
monteCarloSinCRNv1 <- function(params){
  stopifnot(params[2] > 0)
  mean(sin(params[1] + params[2]*commonZs))
}
# monteCarloSinCRNv1(c(10,1))
```

Let's compare using common random numbers to going without. As you can see, common random numbers make the plot look "smoother."

```r
mcApproxCRNv1 <- matrix(apply(muSigmaGrid, 1, monteCarloSinCRNv1), ncol = leng
par(mfrow=c(1,2))
contour(muGrid, sigmaGrid, mcApprox, xlab = "mu", ylab = "sigma", main = "mc w
contour(muGrid, sigmaGrid, mcApproxCRNv1, xlab = "mu", ylab = "sigma", main =
```

r_and_python_book_files/figure-latex/unnamed-chunk-

```r
par(mfrow=c(1,1))
```

The downside to this implementation is that we have a bunch of samples floating around in the global environment. We can implement this much more nicely with a function factory.

```r
makeMCFunc <- function(){
  commonZs <- rnorm(n=n)
  function(params){
    stopifnot(params[2] > 0)
    mean(sin(params[1] + params[2]*commonZs))
  }
}
monteCarloSinCRNv2 <- makeMCFunc()
# monteCarloSinCRNv2(c(10,1))
```

Much better! Let's just make sure this function works by comparing its output to the known true function.

`r_and_python_book_files/figure-latex/unnamed-chunk-242-1.pd`

5.8 Functions as Function Outputs in Python

Returning functions from functions, explaining mechanics

`np.vectorize` is something that I use quite often.

decorators!

TODO Both R and Python have reduce functions. R has `Reduce`, while Python has `reduce`.

In R

reduce,

5.8.1 Exercises

1. Let X_t be the price of a stock at time t. A *call (or put) option* is a type of derivative contract you can buy or sell. It's called a derivative because its value derives from the value of a stock. A single call contract (or put) gives the owner the right to buy (or sell) stock (100 shares) at a certain price. This certain price written into an options contract

(s) is called the *strike price*. The day the option expires is called the *expiration date*. If a holder of the option uses his/her right to buy or sell the underlying stock, he/she is said to have *exercised* the option.

The price and value of options floats around randomly through time just like a stock does. Part of the value of an option is its *intrinsic value*. It is not the same as the overall value of an option, or its "fair price"–rather, the intrinsic value represents how much money its possible to guarantee at the current moment, if you exercised the option, and then liquidated the resulting position in the underlying.

For example, call owners hope the stock rises above the strike price so that they can buy the stock for a low price, and immediately sell it for a high price. If $X_t > s$, they could buy at s, and sell at X_t. In other words, the intrinsic value of a call option is

$$C_t^s = 100 \times \max(X_t - s, 0)$$

as long as t is before the expiration date. If $X_t < s$, then the owner doesn't have to buy the stock, so the call at that time is intrinsically worthless.

Put owners hope the stock price falls below the strike price, because a put's intrinsic value is

$$P_t^s = 100 \times \max(s - X_t, 0)$$

Sometimes people buy and sell multiple contracts (with the same expiration date) at the same time. If they do, the intrinsic value of the entire position is additive. For example, if you bought two of the same put option as the one above, then you just double the above expression to get the intrinsic value.

As another example, if you buy (you can also sell) a *call spread*, you buy one call, and then sell another call with the same expiration and a higher strike price. Assuming $s_1 < s_2$ the intrinsic value is of the call spread with strikes s_1 and s_2 is

$$100 \times \max(X_t - s_1, 0) - 100 \times \max(X_t - s_2, 0)$$

a. In R, write a function called `getIntrinsicValueFunc` that takes three vectors as arguments, (`dirs`, `strikes`, `contractType`, `numContracts`), and returns a function. These arguments should all be the same length. The length of these inputs is the number of contracts in an options position. The function that `getIntrinsicValueFunc` returns should only take one argument: `prices`, a vector of prices of the underlying stock. The function that `getIntrinsicValueFunc` should return a vector of intrinsic values over time.

Because the arguments may only take on certain values, and because the function to be returned has a certain signature, a template has been provided below.

```
getIntrinsicValueFunc <-function(dirs, strikes, contractType, numContracts){
  stopifnot(all(dirs %in% c("buy", "sell")))
  stopifnot(all(contractType %in% c("call", "put")))
  stopifnot(all(is.integer(strikes)))
  stopifnot(all(is.integer(numContracts)))
  stopifnot( (length(dirs) == length(strikes)) && (length(dirs) == length(contractTyp

  usefulFunc <- function(prices){
    # TODO
  }
  return(usefulFunc)
}
```

b. Create an intrinsic value plot. The x-axis should be a grid of values: let's say from 100 to 200. On the y-axis plot the intrinsic value of buying a call spread with strikes 110 and 120.

c. Do the same thing as above but show what happens when you sell the same call spread.

d. Plot the intrinsic value of a bought put spread with strikes 110 and 120.

e. Plot the intrinsic value of a sold put spread with strikes 110 and 120. What is the difference between all four plots?

2. Do the same thing, but in Python!

creating row masks in a data frame

.apply with weird predicate function

Optimization example

Importance sampling example

bootstrap

numerical integration

3. verbal, open-ended wuestion about what happens when you make a typo in a variable name? R versus Python?
4. try to think of the most complicated program that uses two functions that both use (and modify) globals

Bibliography

Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition.

Brown, T. (2021). *cPseudoMaRg: Constructs a Correlated Pseudo-Marginal Sampler*. R package version 1.0.0.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, USA, 1st edition.

Robert, C. P. and Casella, G. (2005). *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag, Berlin, Heidelberg.

VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, Inc., 1st edition.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wilkinson, L. (2005). *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg.