**Learn Python for Economic Computation**

**by**

**James Caton**

**Cameron Harwick**

**Working Draft – Do Not Distribute**

**Table of Contents**

**[Introduction 3](#Introduction)**

[**Chapter 1: The Essentials 5**](#TheEssentials)

[**Chapter 2: Working with Lists 15**](#WorkingWithLists)

[**Chapter 3: Building Functions 30**](#BuildingFunctions)

[**Chapter 4: Classes and Methods 41**](#ClassesAndMethods)

[**Chapter 5: Working With Numpy and Pandas 48**](#WorkingWithNumpyAndPandas)

[**Chapter 6: Importing, Cleaning, and Analyzing Data 68**](#ImportingCleaningAndAnalyzingData)

[**Chapter 7: Building an OLS Regression Function 75**](#BuildingAnOLSRegressionFunction)

[**Chapter 8: Agent-based Models 96**](#AgentBasedModels)

**[Introduction: Learn Python for Economic Computation](#TableOfContents)**

More than ever, programming skills are essential for professional success. This is true whether your work is in business or in academia. Further, programming is often not enough. You must also be able to work with statistical software or libraries. While many books exist to teach you how to program, there are none that I know of that do a good job of providing training in statistic and econometrics while also providing good programming habits.

**The Beginning**

I first began programming in 2014. I wanted to have a strong grasp of the fundamental pieces of programming before I moved on to complex structures. *I did not want to learn to use libraries without understanding the fundamentals of* *Python*. I started out with Python, but since I was interested in agent-based modeling, I spent more time working with NetLogo. The use of objects in NetLogo is limited in comparison to languages like Java and Python. Still, it provided a good starting point to learn the basics of scripting, of functions (methods), and the use and interaction of objects. With the help of a friend and many hours spent exploring Python with small projects, I grew comfortable with the language. The building blocks I learned in the process will be used to help you grow comfortable with Python as well.

In what follows, you will learn how to install Python quickly and easily. You will learn how to work with objects and functions that are essential to data management and statistical programming. These include working with basic math functions, lists, dictionaries, tuples, if-statements, "for-loops", classes, and reading and writing files in Python. As we develop understanding of the core interface, we will also build a statistical packages.

**Why Python?**

You may wonder why you should learn python as compared to R or MATLAB. Although R and MATLAB are both powerful, they are not well-suited for general purpose programming in the way that Java or C++ are. Python does meet this criterion. Python is intuitive, not differing as much as R from the syntax and structure of traditional programming languages. If you learn Python for data analysis, you

will also have a head-start on programming for other purposes like web design, natural language processing, graphical user interfaces, and so on. For this reason, Python is more useful more generally. By the time you finish this book, you will be well prepared to develop a practice that includes a broad set of applications.

**Installation**

In this book, we will use Python 3. All examples will be generated from Spyder, the Integrated Development Environment provided in Anaconda. Download the Anaconda package at:

https://www.continuum.io/downloads

Download the latest installer for Python 3. If you know your system is 64-bit, choose the 64-bit installer. Otherwise, choose the 32-bit installer. If you have not installed Python on your system before, check the option “Add Anaconda to the system Path environment variable. This option is “Not recommended” by the installation as it gives Anaconda preference over previously installed software. For our purposes, this should not be a problem. Once you have completed the installation, if you would like to install any package, open the command line (in Windows this is PowerShell) and type *conda install package-name* where *package-name* is replaced with the package you wish to install. For example, you can install numpy with *conda install numpy*. If a package is not available in Anaconda, you can install it using *pip install package-name*. The latest version of Anaconda will recognize a library installed in this manner. It is sometimes possible to install unofficial releases of packages with Anaconda, but this will not be necessary for the material covered. Unless otherwise specified, the packages we will be using are included in with the Anaconda installation.

When you are ready to build your first program, open Spyder. You can find it by using the search function for your operating system. I recommend that you place it somewhere on your desktop or pin it to your taskbar.

**Python Scripts**

As you follow along with the book, you may prefer to view the *.py* scripts in your text editor. All scripts can be found at the github for this book: <https://github.com/jlcatonjr/Learn-Python-for-Stats-and-Econ>.

**[Chapter 1: The Essentials](#TableOfContents)**

**a. Printing**

Printing is a feature that you will continually use while you program. It is also useful for creating markers in your code when you are not sure what is wrong. You can print plain text. Using Spyder, we will create a file called helloWorld.py.

1. #helloWorld.py
2. **print**("Hello world!")

To execute this file, select the *Run* button in Spyder or press F5.

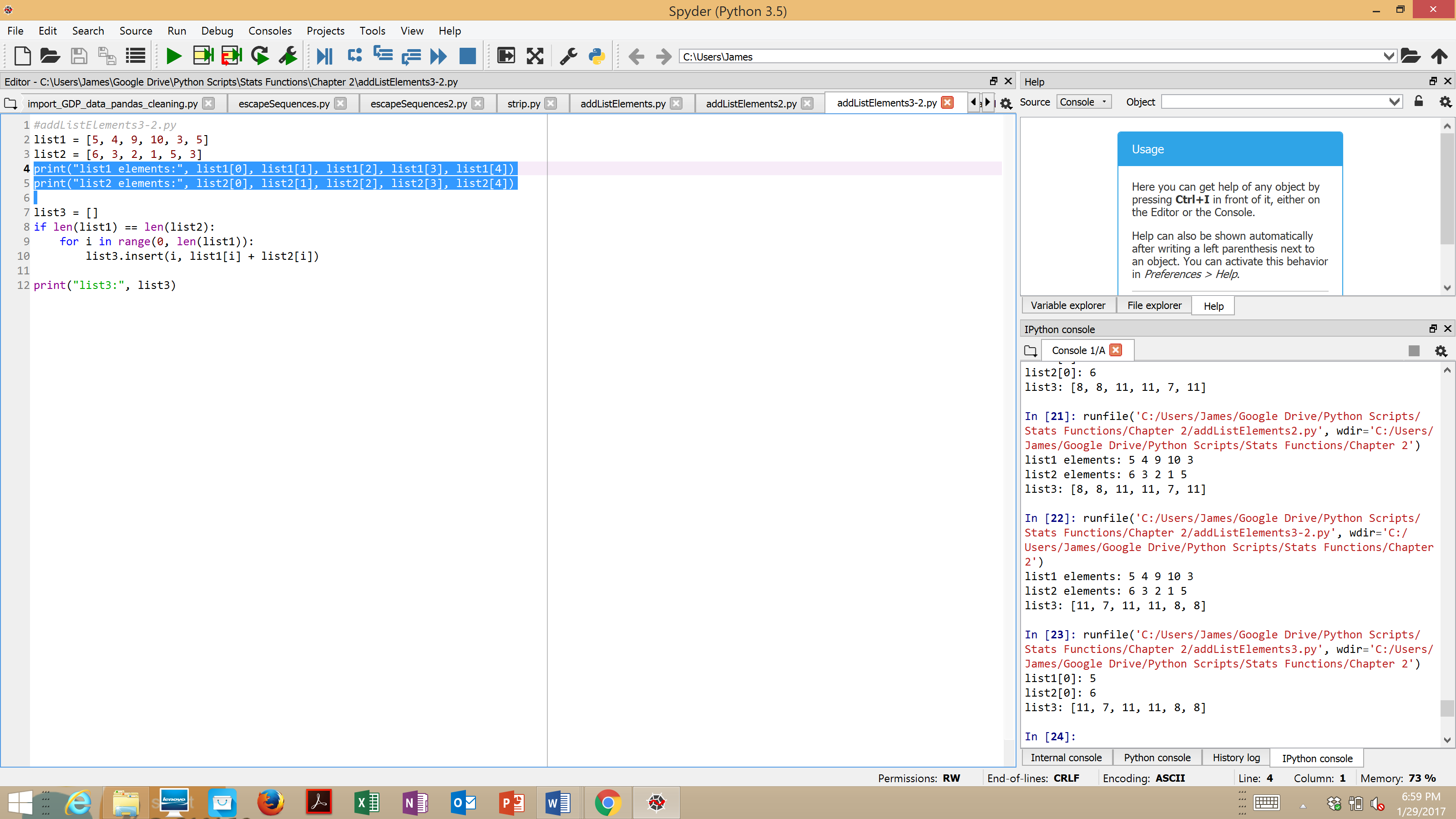


Figure 1 – *Run File* Button

This will automatically ask you to save the file if you have not already saved it. Name the file h*elloWorld.py*.

Output:

Hello world!

You have just created your first program! There is still much to learn.

Let's break down this program. You used the function *print()*. Any string you want to print must be in quotations. Single or double quotes will do, as long as the same type of quote is used at the beginning and the end of the string. Below we run the same program with single quotation marks.

1. #helloWorldSingleQuote.py
2. **print**(‘Hello world!’)

Output:

Hello world!

**b. Create a String Object**

We printed a sentence directly, but what if we want to define the phrase before

we print it? Follow the code below and create an object whose name is *msg*.

*msg* points to a string *“Hello Jim!”*

1. #helloJim.py
3. msg = "Hello Jim!"
4. **print**(msg)

Output:

Hello Jim!

**c. String Functions**

Python makes manipulating strings easy. There are some essential features that you should be aware of and refer back to whenever you need them. We will use several of these in the next file

1. #stringCaps.py
2. msg = "john nash"
3. **print**(msg)
4. **print**(msg.upper())
5. **print**(msg.title())

Output:

john nash

JOHN NASH

John Nash

We used some functions that are owned by strings in python. The function *upper()* makes all strings upper case and *title()* capitalizes the first letter of each word.

We may also create multiple string objects and concatenate them. When we concatenate strings, we join them together.

1. #concatenateStrings.py
2. msg = "john" + " " + "nash"
3. **print**(msg)

Output:

john nash

We can also concatenate string objects that have already been created by using the name that points to the objects.

1. #concatenateStrings2.py
2. line1 = "You thought it would be easy"
3. line2 = "You thought it wouldn't be strange"
4. line3 = "But then you started coding"
5. line4 = "Things never were the same"
6. **print**(line1 + line2 + line3 + line4)

Notice that if we print the lines together, there will be no spaces between them.

Output:

You thought it would be easyYou thought it wouldn't be strangeBut then you started codingThings never were the same

There are several approaches we take to fix this problem. We will try two different approaches. First we will add spaces.

1. #concatenateStringsSpaces.py
2. line1 = "You thought it would be easy"
3. line2 = "You thought it wouldn't be strange"
4. line3 = "But then you started coding"
5. line4 = "Things never were the same"
6. **print**(line1 + " " + line2 + " " + line3 + " " + line4)

Output:

You thought it would be easy You thought it wouldn't be strange But then you started coding Things never were the same

We will achieve the same output if we insert commas between objects in the print fucntion

1. #concatenateStringsCommas.py
2. line1 = "You thought it would be easy"
3. line2 = "You thought it wouldn't be strange"
4. line3 = "But then you started coding"
5. line4 = "Things never were the same"
6. **print**(line1, line2, line3, line4)

Instead of spaces, we can have output return a new line for each string by using the character '\n'. The use of the '\' tells the program that it should read the next character as referring to a special function for formatting.

1. #concatenateStringsNewLines.py
2. line1 = "You thought it would be easy"
3. line2 = "You thought it wouldn't be strange"
4. line3 = "But then you started coding"
5. line4 = "Things never were the same"
6. **print**(line1 + "\n" + line2 + "\n" + line3 + "\n" + line4)

Output:

You thought it would be easy

You thought it wouldn't be strange

But then you started coding

Things never were the same

Python also allows for quotes to be used within quotes. If a string a surrounded by double quotes, then single quotes may be used within that string. Likewise, double quotes may also be used within a string surrounded by single quotes.

1. #quotesInQuotes.py
2. single\_in\_double = "We may use 'single quotes' within double quotes"
3. double\_in\_single = 'We may use "double quotes" within single quotes'
4. **print**(single\_in\_double)
5. **print**(double\_in\_single)

Output:

We may use 'single quotes' within double quotes

We may use "double quotes" within single quotes

Below is a list of related commands. These are known as 'escape sequences'.

**d. Escape Sequences**

\\ = \

\' = '

\" = "

\n = moves text that follows to a new line

\t = move text one tab to the right

This list is incomplete, but will suffice for the purpose of learning the basics of programming and statistics. We use them in the escapeSequences.py.

1. #escapeSequences.py
2. single\_quotes = 'We may use \'single quotes\' in single quotes.'
3. double\_quotes = "Or we may use \"double quotes\" in double quotes."
4. read\_backslash = "We may use two backslashes to print a single backslash: \\"
5. lines = "Every\nword\nis\na\nnew\nline"
6. new\_line\_and\_tab = "We may start a new line \n\tand use tab for a hanging indent"
8. **print**(single\_quotes)
9. **print**(double\_quotes)
10. **print**(read\_backslash)
11. **print**(lines)
12. **print**(new\_line\_and\_tab)

Output:

We may use 'single quotes' in single quotes.

Or we may use "double quotes" in double quotes.

We may use two backslashes to print a single backslash: \

Every

word

is

a

new

line

We may start a new line

and use tab for a hanging indent

**e. More String Functions**

Sometimes we may want to transform strings by adding or removing spaces. We can remove space quite easily with the strip() commands.

.lstrip(): remove spaces on the far left

.rstrip(): remove spaces on the far right

.strip(): remove spaces on the left and right of text object

We will use them in the strip.py file.

1. #strip.py
2. spaces = "    Look at all the spaces in the text!   "
3. **print**("no spaces removed:\n", spaces)
5. remove\_left\_spaces = spaces.lstrip()
6. remove\_right\_spaces = spaces.rstrip()
7. remove\_left\_and\_right\_spaces = spaces.strip()
9. **print**("Remove left spaces:\n" + remove\_left\_spaces )
10. **print**("Remove right spaces:\n" + remove\_right\_spaces )
11. **print**("Remove left and right spaces:\n" + remove\_left\_and\_right\_spaces )

Output:

No spaces removed:

Look at all the spaces in the text!

Remove left spaces:

Look at all the spaces in the text!

Remove right spaces:

Look at all the spaces in the text!

Remove left and right spaces:

Look at all the spaces in the text!

Only the spaces on the left or right side of the entire string were removed. There is still space left in the *spaces* string. We can remove all spaces with the replace command. We identify the strings that we will replace in the first placeholder and the string it will be replaced with in the second.

1. #replace.py
2. spaces = "    Look at all the spaces in the text!      "
3. **print**("No spaces removed:\n", spaces)
5. remove\_all\_spaces = spaces.replace(" ", "")
6. **print**("Remove all spaces:\n" + remove\_all\_spaces)

Output:

No spaces removed:

Look at all the spaces in the text!

Remove all spaces:

Lookatallthespacesinthetext!

You may want to print more than one line without entering multiple instances of the print function. To do this we use three quotation marks on either side of the string to be printed.

1. #stringTripleQuotes.py
2. x = """\
3. Everything in this object will be recorded exactly as entered,
4. if we enter a new line or
5. a new line with a tab."""
7. **print**(x)

Output:

Everything will be recorded exactly as entered,

if we enter a new line or

a new line with a tab

**f. Working with numbers**

In many programming languages, you must identify the type of variable that you create. String must be identified by "str" or something similar. Likewise, integers by "int" and floats by "float". Particular formulations will vary, but the same pattern holds. You do not have to worry about this in Python. Depending on the context, numbers will be automatically cast as integers or floats. If a float is not needed, an integer will be used. The following are integers.

1. #integers.py
2. num1 = 5 + 3
3. num2 = 5 - 4
4. num3 = 4 - 3
6. **print**("num1:",num1, "\nnum2:", num2, "\nnum3:", num3)

Output:

num1: 8

num2: 1

num3: 1

Be aware that the addition sign will add numbers and concatenate strings. We can compare the outcomes by transforming *num1* in the above example into a string.

1. #integersVsStrings.py
3. num1 = 5 + 3
4. num1s = "5" + "3"
6. **print**("num1:", num1,"\nnum1s:", num1s)

Output:

num1: 8

num1s: 53

Floats are numbers that include a decimal place. Different floats may have different sizes, but for now it is enough to distinguish between integers and floats.

1. #floats.py
2. num1 = 5 / 3
3. num2 = 5 / 4
4. num3 = 4 / 3
6. **print**("num1:",num1, "\nnum2:", num2, "\nnum3:", num3)

Output:

num1: 1.6666666666666667

num2: 1.25

num3: 1.3333333333333333

We can also choose to create lists with these labels and numbers and print them in entirety. We can also print individually the elements from multiple lists using a for loop. We will discuss for loops more later. For now, it is enough to understand that the command “for i in range(j)” , which can also be written as “for I in range(0, j)”, cycles through the values from 0 to (k – 1). Since an array index starts at zero, the value equal to the length of an array can be used to cycle through all of its elements.

1. #printLists.py
2. num1 = 5 / 3
3. num2 = 5 / 4
4. num3 = 4 / 3
5. num\_list = [num1, num2, num3]
6. num\_label = ["num1:","num2:","num3:"]
7. **print**("We can print the lists in entirety")
8. **print**(num\_label, num\_list)
10. **print**("\nOr we can print the ith element from each list for all j elements")
11. j = len(num\_ist)
12. **for** i **in** range(j):
13. **print**(num\_label[i], num\_list[i])

Output:

We can print the lists in entirety

['num1:', 'num2:', 'num3:'] [1.6666666666666667, 1.25, 1.3333333333333333]

Or we can print the ith element from each list for all j elements

num1: 1.6666666666666667

num2: 1.25

num3: 1.3333333333333333

We will learn more about for-loops in the following chapter.

Remember that we can concatenate strings such that “this string” + “ and that string” = “this string and that string”. Let’s try to concatenate a string and a float from the above example.

1. #printError.py
2. num1 = 5 / 3
3. num2 = 5 / 4
4. num3 = 4 / 3
5. sum\_nums = num1 + num2 + num3
7. **print**("num1 + num2 + num3: " + sum\_nums)

Output:

TypeError: Can't convert 'float' object to str implicitly

The addition symbol will either sum values or concatenate strings. It cannot concatenate a value and a string unless that value is casted as a string. We can fix this by casting sumNums as a string when we print:

1. #castString.py
2. num1 = 5 / 3
3. num2 = 5 / 4
4. num3 = 4 / 3
5. sum\_nums = num1 + num2 + num3
7. **print**("num1 + num2 + num3: " + str(sum\_nums))

Output:

num1 + num2 + num3: 4.25

As you have seen in above examples, using a comma to separate objects passed to the print function will place space between them and allow objects of different types to be passed with the same instance of the print function.

1. #printMixedTypesWithComma.py
2. num1 = 5 / 3
3. num2 = 5 / 4
4. num3 = 4 / 3
5. sum\_nums = num1 + num2 + num3
7. **print**("num1 + num2 + num3:", sumNums)

Output:

num1 + num2 + num3: 4.25

Computers must indicate the amount of space a particular value will hold before assigning the value. Other languages force you to cast your value beforehand. Python assumes that you want to use a double if you include a decimal point. Floats in python are actually doubles with 64-bit precision. We can check this by importing "sys".

1. #checkFloat.py
3. **import** sys
4. **print**(sys.float\_info)

Output:

sys.float\_info(max=1.7976931348623157e+308, max\_exp=1024, max\_10\_exp=308, min=2.2250738585072014e-308, min\_exp=-1021, min\_10\_exp=-307, dig=15, mant\_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

The results show us that floats are large, but that do have a finite size. Remember decimals will be defined as floats. We reach the limit at 2.01023 . In Python we indicate an exponent with \*\*, as in the example below.

1. x = 2.0 \*\* 1023
2. **print**(x)

Output:

8.98846567431158e+307

If we enter "num = 2.0 \*\* 1024", Python will automatically convert the value into an integer.

1. #checkFloat.py
3. **import** sys
4. **print**(sys.float\_info)
6. x = 2.0 \*\* 1023
7. **print**(type(x))
8. **print**(x)
10. y = 2 \*\* 1025
11. **print**(type(y))
12. **print**(y)

Output:

sys.float\_info(max=1.7976931348623157e+308, max\_exp=1024, max\_10\_exp=308, min=2.2250738585072014e-308, min\_exp=-1021, min\_10\_exp=-307, dig=15, mant\_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

<class 'float'>

8.98846567431158e+307

<class 'int'>

359538626972463181545861038157804946723595395788461314546860162315465351611001926265416954644815072042240227759742786715317579537628833244985694861278948248755535786849730970552604439202492188238906165904170011537676301364684925762947826221081654474326701021369172596479894491876959432609670712659248448274432

As we learn to program, we will come upon a number of unique problems. To the extent that we can learn about them systematically we will, otherwise we will deal with them as they come up.

**[Chapter 2: Working With Lists](#TableOfContents)**

Much of the remainder of this book is dedicated to using lists to produce analysis that is low cost and elegant. To use the words of economics, you are making a long-term investment in your human capital. Once you have paid the necessary fixed-costs, you can work with data at low marginal cost.

If you are familiar with other languages, you may be accustomed to working with arrays. The size of an array is static. An array is unable to change its size unless a new object is created in its place. By default, Python works with dynamic lists instead of arrays. This means that you can add elements to a list that already exists without creating a new list object.

**a. Working with Lists**

In later chapters, we will combine lists with dictionaries to essential data structures. We will also work with more efficient and convenient data structures using the numpy and pandas libraries.

Below we make our first lists. One will be empty. Another will contain integers. Another will have floats. Another strings. Another will mix these:

1. #lists.py
3. empty\_list = []
4. int\_list = [1, 2, 3, 4, 5]
5. float\_list = [1.0, 2.0, 3.0, 4.0, 5.0]
6. string\_list = ["Many words", "impoverished meaning"]
7. mixed\_list = [1, 2.0, "Mix it up"]
8. **print**(empty\_list)
9. **print**(int\_list)
10. **print**(float\_list)
11. **print**(string\_list)
12. **print**(mixed\_list)

Output:

[]

[1, 2, 3, 4, 5]

[1.0, 2.0, 3.0, 4.0, 5.0]

['Many words', 'impoverished meaning']

[1, 2.0, 'Mix it up']

Often we will want to transform lists. In the following example, we will concatenate two lists, which means we will join the lists together:

1. #concatenateLists.py
3. list1 = [5, 4, 9, 10, 3, 5]
4. list2 = [6, 3, 2, 1, 5, 3]
5. join\_lists = list1 + list2
7. **print**("list1:", list1)
8. **print**("list2:", list2)
9. **print**("join\_lists:", join\_lists)

Output:

List1: [5, 4, 9, 10, 3, 5]

List2: [6, 3, 2, 1, 5, 3]

join\_lists: [5, 4, 9, 10, 3, 5, 6, 3, 2, 1, 5, 3]

We have joined the lists together to make one long list. We can already observe one way in which Python will be useful for helping us to organize data. If we were doing this in a spread sheet, we would have to identify the row and column values of the elements or copy and paste the desired values into new rows or enter formulas into cells. Python accomplishes this for us with much less work.

We want to be able to access particular elements in a list and to have these elements interact. For a list of numbers, we will usually perform some arithmetic operation or categorize these values in order to identify meaningful subsets within the data.

In the next exercise we will call elements by index number from the same lists we have already made. We will use the list’s *append* method to make a copy of a list. The *append* method adds an element to the end of a list.

1. #copyListElements.py
3. list1 = [5, 4, 9, 10, 3, 5]
4. list2 = [6, 3, 2, 1, 5, 3]
5. **print**(“list1 elements:”, list1[0], list1[1], list1[2], list1[3], list1[4])
6. **print**(“list2 elements:”, list2[0], list2[1], list2[2], list2[3], list2[4])
8. list3 = []
9. list3.append(list1[0])
10. list3.append(list1[1])
11. list3.append(list1[2])
12. list3.append(list1[3])
13. list3.append(list1[4])
14. list3.append(list1[5])
16. **print**(“list3:”, list3)

Output:

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

list3: [5, 4, 9, 10, 3, 5]

**b. For Loops and *range()***

We can use a for loop to more efficiently execute this task. As we saw in the last chapter, the for loop will execute a series of elements: *for element in list*. Often, this list is a range of numbers that represent the index of a dynamic list. For this purpose we call:

*for i in range(j ,k , l)*:

<execute *script>*

The for loop cycles through all integer of interval *l* between *j* and *k-1*, executing a script for each value. This script may explicitly integrate the value *i.*

If you do not specify a starting value, *j*, the *range* function assumes that you are calling an array of elements from 0 to j. Likewise, if you do not specify an interval, *l*, range assumes that this interval is 1. Thus, *for i in range(k)* is interpreted as *for i in range(0, k, 1).* We will again use the loop in its simplest form, cycling through number from 0 to (*j* – 1), where the length of the list is the value *j*. These cases are illustrated below in *range.py*.

1. #range.py
3. list1 = list(range(9))
4. list2 = list(range(-9,9))
5. list3 = list(range(-9,9,3))
7. **print**(list1)
8. **print**(list2)
9. **print**(list3)

Output:

[0, 1, 2, 3, 4, 5, 6, 7, 8]

[-9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8]

[-9, -6, -3, 0, 3, 6]

The for loop will automatically identify the elements contained in range without requiring you to call *list()*. This is illustrated below in *forLoopAndRange.py*.

1. #forLoopAndRange.py
3. **for** i **in** range(10):
4. **print**(i)

Output:

0

1

2

3

4

5

6

7

8

9

Having printed *i* for all *i* in *range(0, 10, 1)*, we produce a set of integers from 0 to 9.

If we were only printing index numbers in a range, for loops would not be very useful. For loops can be used to produce a wide variety of outputs. Often, you will call a for loop to cycle through the index of a particular array. Since arrays are indexed starting with 0 and for loops also assume 0 as an initial value, cycling through a list with a for loop is straight-forward. For a list named *A*, just use the command:

*For i in range(len(A)):*

*<execute script>*

This command will call all integers between 0 and 1 less than the length of *A*. In other words, it will call all indexers associated with *A*.

1. #copyListElementsForLoop.py
3. list1 = [5,4,9,10,3,5]
4. list2 = [6,3,2,1,5,3]
5. **print**("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
6. **print**("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])
7. list3 =[]
8. j = len(list1)
9. **for** i **in** range(j):
10. list3.append(list1[i])
11. k = len(list2)
12. **for** i **in** range(k):
13. list3.append(list2[i])
14. **print**("list3:", list3)

Output:

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

list3 elements: [5, 4, 9, 10, 3, 5, 6, 3, 2, 1, 5, 3]

**c. Creating a New List with Values from Other Lists**

We can extend the exercise by summing the *ith* elements in each list. In the exercise below, *list3* is the sum of the *ith* elements from *list1* and *list2*.

1. #addListElements.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = [6, 3, 2, 1, 5, 3]
4. **print**(“list1 elements:”, list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
5. **print**(“list2 elements:”, list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])
6. list3 = []
7. j = len(list1)
8. **for** i **in** range(j):
9. list3.append(list1[i] + list2[i])
11. **print**("list3:", list3)

Output:

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

list3: [11, 7, 11, 11, 8, 8]

In the last exercise, we created an empty list, *list3*. We could not fill the list by calling element in it directly, as no elements yet exist in the list. Instead, we use the *append* method that is owned by the list-object. Alternately, we can use the *insert* method owned by the list-object. It takes the form, *list.insert(index, object)*. This is shown in a later example. We appended the summed values of the first two lists in the order that the elements are ranked. We could have summed them in opposite order by summing element 5, then 4, ..., then 0.

1. #addListElements2.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = [6, 3, 2, 1, 5, 3]
4. **print**("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4])
5. **print**("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4])
7. list3 = []
8. j = len(list1)
9. **for** i **in** range(j):
10. list3.append(list1[i] + list2[i])
11. **print**("list3:", list3)

Output:

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

list3: [8, 8, 11, 11, 7, 11]

In the next exercise we will us a new function that we have not used before. We will check the length of each list whose elements are summed. We want to make sure that if we call an element in one list, it exists in the other. We do not want to call a list element if it does not exist. That would produce an error.

An if statement checks if a stipulated condition is true. As with the for loop, the if statement is followed by a colon. This tells the program that the execution below or in front of the if statement depends upon the truth of the condition specified. The code that follows below an if statement must be indented, as this identifies what block of code is subject to the statement.

When executed, the program either returns:

*if True:*

*<execute script>*

If the condition is true, then the commands that follow the if-statement will be executed. Though not stated explicitly, we can think of the program as passing over the if statement to the remainder of the script:

*if True:*

*<execute script>*

*else:*

*pass*

We are interested in checking that two lists have the same length. To check that a variable has a stipulated value, we use two equals signs. Using *==* identifies that a we are comparing two values rather setting the value of the variable on the left.

Following the if statement is a for loop. If the length of list1 and list2 are equal, the program will set the *ith* element of list3 equal to the sum of the *ith* elements from list1 and list2. In this example, the for loop will cycle through index values 0, 1, 2, 3, 4, and 5.

We can take advantage of the for loop to use *insert()* in a manner that replicates the effect of our use of *append()*. We will insert the sum of the ith elements of list1 and list2 into the ith element of list3.

1. #addListElements3.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = [6, 3, 2, 1, 5, 3]
4. **print**("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
5. **print**("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])
7. list3 = []
8. j = len(list1)
9. **if** len(list1) == len(list2):
10. **for** i **in** range(0, len(list1)):
11. list3.insert(i, list1[i] + list2[i])
12. **else**:
13. **print**("Lists are not the same length, cannot perform element-wise operations.")
15. **print**("list3:", list3)

Output:

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

list3: [11, 7, 11, 11, 8, 8]

The if condition may be followed by an else statement. This tells the program to run a different command if the condition of the if statement is not met. In this case, we want the program to tell us why the condition was not met. In other cases, you may want to create other if statements to create a tree of possible outcomes. Below we use an if-else statement to identify when list’s are not the same length. We remove the last element from list 2 to create lists of different lengths:

1. #addListElements4.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = [6, 3, 2, 1, 5]
4. **print**("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4])
5. **print**("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4])
7. list3 = []
8. j = len(list1)
9. **if** len(list1) == len(list2):
10. **for** i **in** range(0, len(list1)):
11. list3.insert(i, list1[i] + list2[i])
12. **else**:
13. **print**("Lists are not the same length, cannot perform element-wise operations.")
15. **print**("list3:", list3)

Output:

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

Lists are not the same length, cannot perform element-wise operations.

list3: []

Since the condition passed to the if statement was false, no values were appended to *list3*.

**d. Removing List Elements**

Perhaps you want to remove an element from a list. There are a few means of accomplishing this. Which one you choose depends on the ends desired.

1. #deleteListElements.py
3. list1 = ["red", "blue", "orange", "black", "white", "golden"]
4. list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
5. **print**("lists before deletion: ")
6. **if** len(list1) == len(list2):
7. **for** i **in** range(len(list1)):
8. **print**(list1[i], "       ", list2[i])
10. **del** list1[0]
11. **del** list2[5]
13. **print**("lists after deletion: ")
14. **if** len(list1) == len(list2):
15. **for** i **in** range(len(list1)):
16. **print**(list1[i], "       ", list2[i])

Output:

red nose

blue ice

orange fire

black cat

white mouse

golden dog

lists after deletion:

blue nose

orange ice

black fire

white cat

golden mouse

We have deleted "red" from list1 and "dog" from list2. By printing the elements of each list once before and once after one element is deleted from each, we can note the difference in the lists over time.

What if we knew that we wanted to remove the elements but did not want to check what index each element is associated with? We can use the remove function owned by each list. We will tell list1 to remove "red" and list2 to remove "dog"

1. #removeListElements
3. list1 = ["red", "blue", "orange", "black", "white", "golden"]
4. list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
5. **print**("lists before deletion: ")
6. **if** len(list1) == len(list2):
7. **for** i **in** range(len(list1)):
8. **print**(list1[i], "       ", list2[i])
10. list1.remove("red")
11. list2.remove("dog")
13. **print**("lists after deletion: ")
14. **if** len(list1) == len(list2):
15. **for** i **in** range(len(list1)):
16. **print**(list1[i], "       ", list2[i])

Output:

lists before deletion:

red nose

blue ice

orange fire

black cat

white mouse

golden dog

lists after deletion:

blue nose

orange ice

black fire

white cat

golden mouse

We have achieved the same result using a different means. As the saying goes in programming, “there is more than one way to do it.” What if we wanted to keep track of the element that we removed. Before deleting or removing the element, we could assign the value to a different object. Let's do this before using the remove function:

1. #removeAndSaveListElements
3. list1 = ["red", "blue", "orange", "black", "white", "golden"]
4. list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
5. **print**("lists before deletion: ")
6. **if** len(list1) == len(list2):
7. **for** i **in** range(len(list1)):
8. **print**(list1[i], "\t", list2[i])
10. list1Res = "red"
11. list2Res = "dog"
12. list1.remove(list1\_res)
13. list2.remove(list2\_res)
15. **print**("lists after deletion: ")
16. **if** len(list1) == len(list2):
17. **for** i **in** range(len(list1)):
18. **print**(list1[i], "       ", list2[i])
20. **print**()
21. **print**("Res1\tRes2”
22. **print**(list1Res, “\t”+list2Res)

Output:

lists before deletion:

red nose

blue ice

orange fire

black cat

white mouse

golden dog

lists after deletion:

blue nose

orange ice

black fire

white cat

golden mouse

Res1 Res2

red dog

An easier way to accomplish this is to use *pop*, another method owned by each list.

1. #removeListElementsPop.py
3. # define list1 and list2
4. list1 = ["red", "blue", "orange", "black", "white", "golden"]
5. list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
7. #identify what is printed in for loop
8. **print**("lists before deletion: ")
9. # use for loop to print lists in parallel
10. **for** i **in** range(len(list1)):
11. **print**(list1[i], "\t", list2[i])
13. #remove list elements and save them as variables "\_res"
14. list1\_res = list1.pop(0)
15. list2\_res = list2.pop(5)
17. **print**()
18. # print lists again as in lines 8-11
19. **print**("lists after deletion: ")
20. **for** i **in** range(len(list1)):
21. **print**(list1[i], "\t", list2[i])
22. **print**()
23. **print**("Res1\tRes2")
24. #print elements that were removed from list
25. **print**(list1\_res, "\t", list2\_res)

Output:

lists before deletion:

red nose

blue ice

orange fire

black cat

white mouse

golden dog

lists after deletion:

blue nose

orange ice

black fire

white cat

golden mouse

Res1 Res2

red dog

**e. More For Loops**

When you loop through element values, it is not necessary that these are consecutive. You may skip values at some interval. The next example returns to the earlier *addListElements* examples. This time, we add the number 2 to the end of the for statement. Now *range* will count by twos from *0* to *j – 1*. This will make *list3* shorter than before.

1. #addListElements5.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = [6, 3, 2, 1, 5, 3]
4. **print**("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
5. **print**("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])
7. list3 = []
8. j = len(list1)
9. **if** j == len(list2):
10. **for** i **in** range(0, j, 2):
11. list3.append(list1[i] + list2[i])
12. **else**:
13. **print**("Lists are not the same length, cannot perform element-wise operations.")
14. **print**("list3:", list3)

Output

list1 elements: 5 4 9 10 3

list2 elements: 6 3 2 1 5

list3: [11, 11, 8]

We entered the sum of elements 0, 2, and 4 from lists 1 and 2 into list 3. Since these were appended to list 3, they are indexed in list3[0], list3[1], and list3[2].

You may use a for loop that calls each element in the list without identifying its indexer. This takes the form:

*for x in obj:*

*<execute script>*

Each x called is an element from obj. Where before we passed *len(list1)* to the for loop, we now pass *list1* itself to the for loop and append each element, *x*, to *list2*.

1. #forLoopWithoutIndexer.py
2. list1 = ["red", "blue", "orange", "black", "white", "golden"]
3. list2 = []
4. **for** x **in** list1:
5. list2.append(x)
7. **print**("list1", "\tlist2")
9. **if** len(list1) == len(list2):
10. **for** i **in** range(0, len(list1)):
11. **print**(list1[i], "\t", list2[i])

Output:

list1 list2

red red

blue blue

orange orange

black black

white white

golden golden

**f. Sorting Lists, Errors, and Exceptions**

We can sort lists using the sorted list function that orders the list either by number or alphabetically. We reuse lists from the last examples to show this.

1. #sorting.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = ["red", "blue", "orange", "black", "white", "golden"]
5. **print**("list1:", list1)
6. **print**("list2:", list2)
8. sorted\_list1 = sorted(list1)
9. sorted\_list2 = sorted(list2)
11. **print**("sorted\_list1:", sorted\_list1)
12. **print**("sorted\_list2:", sorted\_list2)

Output:

list1: [5, 4, 9, 10, 3, 5]

list2: ['red', 'blue', 'orange', 'black', 'white', 'golden']

sortedList1: [3, 4, 5, 5, 9, 10]

sortedList2: ['black', 'blue', 'golden', 'orange', 'red', 'white']

What happens if we try to sort a that has both strings and integers? You might expect that Python would sort integers and then strings or vice versa. If you try this, you will raise an error:

1. #sortingError.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = ["red", "blue", "orange", "black", "white", "golden"]
4. list3 = list1 + list2
6. **print**("list1:", list1)
7. **print**("list2:", list2)
8. **print**("list3:", list3)
10. sortedList1 = sorted(list1)
11. sortedList2 = sorted(list2)
12. sortedList3 = sorted(list3)
14. **print**("sortedList1:", sortedList1)
15. **print**("sortedList2:", sortedList2)
16. **print**("sortedList3:", sortedList3)

This returns the following error:

TypeError: '<' not supported between instances of 'str' and 'int'

If this error is raised during execution, it will interrupt the program. One way to deal with this is to ask Python to *try* to execute some script and to execute some other command if an error would normally be raised:

1. #sortingError.py
2. list1 = [5, 4, 9, 10, 3, 5]
3. list2 = ["red", "blue", "orange", "black", "white", "golden"]
4. list3 = list1 + list2
6. **print**("list1:", list1)
7. **print**("list2:", list2)
8. **print**("list3:", list3)
10. sortedList1 = sorted(list1)
11. sortedList2 = sorted(list2)
13. **print**("sortedList1:", sortedList1)
14. **print**("sortedList2:", sortedList2)
15. **try**:
16. sortedList3 = sorted(list3)
17. **print**("sortedList3:", sortedList3)
18. **except**:
19. ("TypeError: unorderable types: str() < int()"
20. "Ignoring error")
22. **print**("Execution complete!")

Output:

list1: [5, 4, 9, 10, 3, 5]

list2: ['red', 'blue', 'orange', 'black', 'white', 'golden']

list3: [5, 4, 9, 10, 3, 5, 'red', 'blue', 'orange', 'black', 'white', 'golden']

sortedList1: [3, 4, 5, 5, 9, 10]

sortedList2: ['black', 'blue', 'golden', 'orange', 'red', 'white']

Execution complete!

We successfully avoided the error and instead called an alternate operation defined under *except*. The use for this will become more obvious as we move along. We will *except* use them from time to time and note the reason when we do.

**g. Slicing a List**

Sometimes, we may want to access several elements instantly. Python allows us to do this with a slice. Technically, when you call a list in its entirety, you take a slice whose size is the whole list. We can do this explicitly like this:

1. #fullSlice.py
2. some\_list = [3,1,5,6,1]
3. **print**(some\_list[:])

Output:

[3, 1, 5, 6, 1]

Using *some\_list[:]* is equivalent of creating a slice using someList[minIndex: listLength] where minIndex = 0 and listLength= len(someList):

1. #fullSlice2.py
2. Some\_list = [3, 1, 5, 6, 1]
3. min\_index = 0
4. max\_index = len(some\_list)
5. **print**("minimum:", min\_index)
6. **print**("maximum:", max\_index)
7. **print**("Full list using slice", some\_list[min\_index:max\_index])
8. **print**("Full list without slice", some\_list)

Output:

minimum: 0

maximum: 5

Full list using slice [3, 1, 5, 6, 1]

Full list without slice [3, 1, 5, 6, 1]

This is not very useful if we do not use this to take a smaller subsection of a list. Below, we create a new array that is a subset of the original array. As you might expect by now, *fullList[7]* calls the 8th element. Since indexing begins with the 0th element, this element is actually counted as the 7th element. Also, similar to the command *for i in range(3, 7)*, the slice calls elements 3, 4, 5, and 6:

1. #partialSlice.py
2. min\_index = 3
3. max\_index = 7
4. full\_list = [1,2,3,4,5,6,7,8,9]
5. partial\_list = fullList[min\_index:max\_index]
6. **print**(full\_list)
7. **print**(partial\_list)
9. **print**("full\_list[7]:", full\_list[7])

Output:

[1, 2, 3, 4, 5, 6, 7, 8, 9]

[4, 5, 6, 7]

fullList[7]: 8

**h. Lists, Lists, and More Lists**

Lists have some convenient features. You can find the maximum and minimum values in a list with the min() and max() functions:

1. #minMaxFunctions.py
3. list1 = [20,30,40,50]
4. max\_list\_value = max(list1)
5. min\_list\_value = min(list1)
6. **print**("maximum:", max\_list\_value, "minimum:", min\_list\_value)

Output:

max: 50 min: 20

We could have used a for loop to find these values. The program below performs the same task:

1. #minMaxFunctionsByHand.py
3. list1 = [20, 30, 40, 50]
5. ### initial smallest value is very high
6. ### will be replaced if a value from the list is lower
7. min\_list\_value = 2 \*\* 1023
9. ### initial largest value is very low
10. ### will be replaced if a value from the list is higher
11. max\_list\_value = -2 \*\* 1023
13. **for** x **in** list1:
14. **if** x < min\_list\_value:
15. min\_list\_value = x
16. **if** x > max\_list\_value:
17. max\_list\_value = x
19. **print**("maximum:", max\_list\_value, "minimum:", min\_list\_value)

Output:

max: 50 min: 20

We chose to make the starting value of min\_list\_value large and positive and the starting value of max\_list\_value large and negative. The for loop cycles through these values and assigns the value, *x*, from the list to min\_list\_value if the value is less than the current value assigned to min\_list\_value and to max\_list\_value if the value is greater than the current value assigned to max\_list\_value.

Earlier in the chapter, we constructed lists using list comprehension (i.e., the *list()* function) and by generating lists and setting values with *.append()* and *.insert()*. We may also use a generator to create a list. Generators are convenient as they provide a compact means of creating a list that is easier to interpret. They follow the same format as the *list()* function.

1. #listFromGenerator.py
3. generator = (i **for** i **in** range(20))
4. **print**(generator)
6. list1 = list(generator)
7. **print**(list1)
9. list2 = [2 \* i **for** i **in** range(20)]
10. **print**(list2)

Output:

<generator object <genexpr> at 0x000002AF760A4FC0>

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]

**[Chapter 3: Building Functions](#TableOfContents)**

Often, when we are performing statistical operations we perform these over a large dataset. It can be difficult to understand the meaning conveyed by these measures. Learning to program presents an opportunity to better understand how functions work. In this chapter we will create some basic statistical functions and compare their output to the functions built into python. By creating the function, you will understand the meaning of summation signs. Computing these statistics by hand would be a laborious process and expensive in terms of time. Once a function is constructed, it can be employed to calculate statistics in a fraction of the time.

**a. Building a Function**

So far, we have built programs on the fly. For purposes of pedagogy, this is fine. As you develop your skills, you want to form good practices. This includes the building of functions for repeated use as well as the building of classes. This chapter we will concentrate on functions. Build all of your functions in the same file, *statsFunctions.py*.

In Python, functions take the form

1. **def** function-name(object1, object2, ...):
2. <operations(objects)>

If the function allows, you will pass an object by calling it in the parentheses that follow the function name. The first function that we build will be the *total()* function. We define the function algebraically as the sum of all values in a list of length j:

Since lists indices start with the integer 0, we will write our functions as starting with i = 0 and process elements to the index of value n-1. Since the range function in Python automatically counts to one less than the value identified, the for-loop used will take the form:

1. **for** values **in** range(n):
2. <operations>

We will use it to return the sum of values in a list. After building this, we will pass a list to the function:

1. #statsFunctions1.py
2. **def** total(list\_obj):
3. total = 0
4. n = len(list\_obj)
5. **for** i **in** range(n):
6. total += list\_obj[i]
7. **return** total
9. list1 = [3, 6, 9, 12, 15]
10. total\_list1 = total(list1)
11. **print**(total\_list1)

Output:

45

The *total()* function is a simple function that will be used in many of the other functions that we write. You can find this and other functions from this chapter in the statsFunctions.py file at <https://github.com/jlcatonjr/Learn-Python-for-Stats-and-Econ/tree/master/Chapter%203>.

**b. Statistical Functions**

**i. Average Statistics**

We define the mean of a set of numbers

The top part of the function is the same as the notation that represents the sum of a list of numbers. Thus, in *mean()*, we call *total()* and divide the result by the length of the list. Then, we use the function to calculate value and save that value as an object

1. #statsFunctions2.py
2. **. . .**
3. **def** mean(list\_obj):
4. n = len(list\_obj)
5. mean = total(list\_obj) / n
6. **return** mean
7. . . .
8. mean\_list1 = mean(list1)
9. **print**("Mean of list1:", mean\_list1)

Output:

45

Mean of list1: 9.0

Now that we have set up total and mean functions, we are ready to calculate

other core statistical values:

1. median

2. mode

3. variance

4. standard deviation

5. covariance

6. correlation

Statistical values provide information about the shape and structure of data. These values are aggregates as they sum some characteristic from the dataset, and transform it to a value representative of the whole dataset. Above, we have already calculated the mean, now we shall calculate the other average values, median and mode.

The median is defined is the middle most number in a list. In a list of odd length, this is straightforward to find. We divide the length of the list plus one by two. To identify if a list is odd or even, we divide the list by 2 using the % sign. This will call the remainder. If the remainder does not equal (!=) zero, then the list of odd length. If the remainder is 0, then the list is of even length. If the list is of even length, we take the average of the two middle terms.

1. #statsFunctions3.py
2. **. . .**
4. **def** median(list\_obj):
5. median = 0
6. **if** len(list\_obj) % 2 != 0:
7. index = int((len(list\_obj)) / 2)
8. median = float(list\_obj[index])
9. **else**:
10. index1 = int((len(list\_obj)) / 2)
11. index2 = index1 - 1
12. median = (list\_obj[index1] + list\_obj[index2]) / 2
13. **return** median
14. . . .
16. list2 = [1, 1, 1, 1, 1, 5]
17. median\_list1 = median(list1)
18. median\_list2 = median(list2)
19. **print**("Median of list1:", median\_list1)
20. **print**("Median of list2:", median\_list2)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

The mode of a list is defined as the number that appears the most in the list. In order to quickly and cleanly identify the mode, we are going to use a new data structure: the dictionary. The dictionary is like a list, but elements are called by a key, not by elements from an ordered set of index numbers. We are going to use the values from the list passed to the function as keys. Every time a value is passed, the dictionary will indicate that it has appeared an additional time by adding one to the value pointed to by the key. We will pass the lists that we used in the previous exercises.

1. #statsFunctions4.py
2. . . .
3. def mode(list\_obj):
4. max\_count = 0
5. counter\_dict = {}
6. **for** value in list\_obj:
7. counter\_dict[value] = 0
8. **for** value in list\_obj:
9. counter\_dict[value] +=1
10. count\_list = list(counter\_dict.values())
11. max\_count = max(count\_list)
12. keys = [key **for** key in counter\_dict **if** counter\_dict[key] == max\_count]
13. mode = []
14. **for** key in keys:
15. mode.append(key)
16. **return** mode
17. . . .
19. mode\_list1 = mode(list1)
20. mode\_list2 = mode(list2)
21. **print**("Mode of list1:", mode\_list1)
22. **print**("Mode of list2:", mode\_list2)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Note that instead of using the command *for i in range(n)*, the command *for value in list\_obj* is used. The first command counts from *0* to *j* using *i* and can be used to call elements in the list of interest by passing *i* in the form *list\_obj[i]*. In the cases above, we called the values directly, passing them to *counter\_dict* to count the number of times each value appears in a list, first initializing the dictionary by setting to *0* the value linked to each key. Then we add *1* for each time that a value appears. We identify that maximum number of times a value appears by taking the maximum value in *count\_list*, which is simply a list of the values held by *counter\_dict*.Once the *count\_list* is created, identify the maximum value of the list and collect keys that point to that value in *counter\_dict* by comparing each the value linked to each key to the *max\_count*.

**ii. Statistics Describing Distribution**

Average values do not provide a robust description of the data. An average does not tell us the shape of a distribution. In this section, we will build functions to calculate statistics describing distribution of variables and their relationships. The first of these is the variance of a list of numbers.

We define population variance as

When we are dealing with a sample, which is a subset of a population of values, then we divide by (n - 1) to unbias the calculation.

We will first build functions that calculate a population's variance and standard deviation. We will then provide an option for calculating sample variance and standard deviation.

1. #statsFunctions5.py
2. **. . .**
4. **def** variance(list\_obj):
5. list\_mean = mean(list\_obj)
6. n = len(list\_obj)
7. sum\_sq\_diff = 0
8. **for** x **in** list\_obj:
9. sum\_sq\_diff += (x - list\_mean) \*\* 2
10. variance = sum\_sq\_diff / n
11. **return** variance
12. . . .
13. variance\_list1 = variance(list1)
14. variance\_list2 = variance(list2)
15. **print**("Variance of list1:", variance\_list1)
16. **print**("Variance of list2:", variance\_list2)

Output:

Variance of list1: 2.0

From a list’s variance, we also calculate its standard deviation as the square root of the variance

This is true for both the population and sample standard deviations. The function looks like

1. #statsFunctions6.py
2. **. . .**
3. **def** SD(list\_obj):
4. SD = variance(list\_obj) \*\* 1/2
5. **return** SD
6. . . .
8. SD\_list1 = SD(list1)
9. SD\_list2 = SD(list2)
10. **print**("Standard deviation of list1:", SD\_list1)
11. **print**("Standard deviation of list2:", SD\_list2)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Variance of list1: 18.0

Variance of list2: 2.222222222222222

Standard deviation of list1: 9.0

Standard deviation of list2: 1.111111111111111

We must account for whether the list is a sample or represents the population. In order to allow for this, we add an option for taking the variance and standard deviation of a sample:

1. #statsFunctions7.py
2. **. . .**
4. **def** variance(list\_obj, sample = False):
5. list\_mean = mean(list\_obj)
6. n = len(list\_obj)
7. sum\_sq\_diff = 0
8. **for** x **in** list\_obj:
9. sum\_sq\_diff += (x - list\_mean) \*\* 2
10. **if** sample == False:
11. variance = sum\_sq\_diff / n
12. **else**:
13. variance = sum\_sq\_diff / ( n-1)
14. **return** variance
16. **def** SD(list\_obj, sample = False):
17. **if** sample == False:
18. SD = variance(list\_obj) \*\* 1/2
19. **else**:
20. SD = variance(list\_obj,sample = True) \*\* (1/2)
21. **. . .**
23. sample\_SD\_list1 = SD(list1, sample = True)
24. sample\_SD\_list2 = SD(list2, sample = True)
25. **print**("Standard deviation of list1 as sample:", sample\_SD\_list1)
26. **print**("Standard deviation of list2 as sample:", sample\_SD\_list2)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Variance of list1: 18.0

Variance of list2: 2.222222222222222

Standard deviation of list1: 9.0

Standard deviation of list2: 1.111111111111111

Standard deviation of list1 as sample: 4.743416490252569

Standard deviation of list2 as sample: 1.632993161855452

We have left to build function for covariance and, correlation, skewness and kurtosis. Covariance measures the average relationship between two variables. Correlation normalizes the covariance statistic a fraction between 0 and 1.

To calculate covariance, we multiply the sum of the product of the difference between the observed value and the mean of each list for value i = 1 through n = number of observations:

We pass two lists through the covariance() function. As with the variance() and stdev() functions, we can take the sample-covariance or the population-covariance.

1. #statsFunctions8.py
2. **. . .**
3. **def** covariance(list\_obj1, list\_obj2, sample = False):
4. mean1 = mean(list\_obj1)
5. mean2 = mean(list\_obj2)
6. cov = 0
7. n1 = len(list\_obj1)
8. n2= len(list\_obj2)
9. **if** n1 == n2:
10. **for** i **in** range(n1):
11. cov += (list\_obj1[i] - mean1) \* (list\_obj2[i] - mean2)
12. **if** sample == False:
13. cov = cov / n1
14. **else**:
15. cov = cov / (n1 - 1)
16. **return** cov
17. **else**:
18. **print**("List observations not equal")
19. **print**("List1 observations:", n1)
20. **print**("List2 observations:", n2)
21. quit()
22. . . .
23. cov\_pop\_list1\_list2 = covariance(list1, list2, sample = False)
24. cov\_sam\_list1\_list2 = covariance(list1, list2, sample = True)
25. **print**("Covariance of population:", cov\_pop\_list1\_list2)
26. **print**("Covariance of sample:", cov\_sam\_list1\_list2)

If you execute this code, the kernel will restart. This will delete any text in the console except for the command prompt. The problem is that the length of *list1* and *list2* are not equal. To fix this, let’s rewrite *list2* with one less *1*. Then, it will have 5 elements instead of 6.

1. list2 = [1,1,1,1,5]
2. covPopList1List2 = covariance(list1, list2, sample = False)
3. covSamList1List2 = covariance(list1, list2, sample = True)
4. **print**("Covariance of population:", covPopList1List2)
5. **print**("Covariance of sample:", covSamList1List2)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Variance of list1: 18.0

Variance of list2: 2.222222222222222

Standard deviation of list1: 9.0

Standard deviation of list2: 1.111111111111111

Standard deviation of list1 as sample: 4.743416490252569

Standard deviation of list2 as sample: 1.632993161855452

Covariance of population: 4.800000000000001

Covariance of sample: 6.000000000000001

We can transform the covariance into a correlation value by dividing by the product of the standard deviations.

We thus divide the average sum of the product of the errors for each variable by the product standard deviations. This normalizes the covariance, providing an easily interpretable value between 0 and 1. The correlation() function that we build will make use of the covariance() function that we have already constructed as well as the stdev() function.

1. #statsFunctions9.py
2. **. . .**
3. **def** correlation(list\_obj1, list\_obj2):
4. cov = covariance(list\_obj1, list\_obj2)
5. SD1 = SD(list\_obj1)
6. SD2 = SD(list\_obj2)
7. corr = cov / (SD1 \* SD2)
8. **return** corr
9. . . .
11. corr\_list1\_list2 = correlation(list1, list2)
12. **print**("Correlation of list1 and list2:", corr\_list1\_list2)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Variance of list1: 18.0

Variance of list2: 2.222222222222222

Standard deviation of list1: 9.0

Standard deviation of list2: 1.111111111111111

Standard deviation of list1 as sample: 4.743416490252569

Standard deviation of list2 as sample: 1.632993161855452

Covariance of population: 4.800000000000001

Covariance of sample: 6.000000000000001

Correlation of list1 and list2: 0.41666666666666663

**Double Check relationship between type of σ used and type the use of n and n - 1**

Not all distributions are normal, so we need statistics that reflect differences in shapes between distributions.

Skewness is a measure of asymmetry of a population of data about the mean. It is the expected value of the cube of the standard deviation.

Asymmetry in distribution exists due either the existence of long or fat tails. If a tail is long, this means that it contains values that are relatively far from the mean value of the data. If a tail is fat, there exists a greater number of observations whose values are relatively far from the mean than is predicted by a normal distribution. Skewness may sometimes be thought of as the direction which a distribution leans. This can be due to the existence of asymmetric fat tails, long tails, or both. For example, if a distribution includes a long tail on the right side, but is normal otherwise, it is said to have a positive skew. The same can be said of a distribution with a fat right tail. Skewness can be ambiguous concerning the shape of the distribution. If a distribution has a fat right tail and a long left tail that is not fat, it is possible that its skewness will be zero, even though the shape of the distribution is asymmetric.

1. #statsFunctions10.py
2. **. . .**
4. **def** skewness(list\_obj, sample = False):
5. mean\_ = mean(list\_obj)
6. skew = 0
7. n = len(list\_obj)
8. **for** x **in** list\_obj:
9. skew += (x - mean\_) \*\* 3
11. skew = skew / n **if** **not** sample **else** skew / (n - 1)
12. SD\_ = SD(list\_obj, sample)
13. skew = skew / (SD\_ \*\* 3)
14. **return** skew
15. . . .
16. skew\_list1 = skewness(list1)
17. skew\_list2 = skewness(list2)
18. **print**("Skewness of list1:", skew\_list1)
19. **print**("Skewness of list2:", skew\_list2)
21. skew\_list1\_sample = skewness(list1, True)
22. skew\_list2\_sample = skewness(list2, True)
23. **print**("Skewness of list1 as sample:", skew\_list1\_sample)
24. **print**("Skewness of list2 as sample:", skew\_list2\_sample)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Variance of list1: 18.0

Variance of list2: 2.222222222222222

Standard deviation of list1: 9.0

Standard deviation of list2: 1.111111111111111

Standard deviation of list1 as sample: 4.743416490252569

Standard deviation of list2 as sample: 1.632993161855452

Covariance of population: 4.800000000000001

Covariance of sample: 6.000000000000001

Correlation of list1 and list2: 0.41666666666666663

Skewness of list1: 0.0

Skewness of list2: 2.9296874999999987

Skewness of list1 as sample: 0.0

Skewness of list2 as sample: 1.3416407864998736

Kurtosis is an absolute measure of the weight of outliers. While in many cases skewness describes the ‘lean’ of a distribution, kurtosis describes the weight of a distribution that is held in the tails. Kurtosis is the sum of the standard deviation of each observation raised to the fourth power. As with the other statistical values, kurtosis can be taken for a population and for a sample.

If an observation is less than one standard deviation from the mean, its value will be relatively insignificant compared to in observation that is relatively farther from the mean.

1. #statsFunctions11.py
2. **. . .**
4. **def** kurtosis(list\_obj, sample = False):
5. mean\_ = mean(list\_obj)
6. kurt = 0
7. n = len(list\_obj)
8. **for** x **in** list\_obj:
9. kurt += (x - mean\_) \*\* 4
10. SD\_ = SD(list\_obj, sample)
11. kurt = kurt / n **if** **not** sample **else** kurt / (n - 1)
12. kurt = kurt / (SD\_ \*\* 4)
14. **return** kurt
15. . . .
16. kurt\_list1 = kurtosis(list1)
17. kurt\_list2 = kurtosis(list2)
18. **print**("Kurtosis of list1:", kurt\_list1)
19. **print**("Kurtosis of list2:", kurt\_list2)
21. kurt\_list1\_sample = kurtosis(list1, True)
22. kurt\_list2\_sample = kurtosis(list2, True)
23. **print**("Kurtosis of list1 as sample:", kurt\_list1\_sample)
24. **print**("Kurtosis of list2 as sample:", kurt\_list2\_sample)

Output:

45

Mean of list1: 9.0

Median of list1: 9.0

Median of list2: 1.0

Mode of list1: [3, 6, 9, 12, 15]

Mode of list2: [1]

Variance of list1: 18.0

Variance of list2: 2.222222222222222

Standard deviation of list1: 9.0

Standard deviation of list2: 1.111111111111111

Standard deviation of list1 as sample: 4.743416490252569

Standard deviation of list2 as sample: 1.632993161855452

Covariance of population: 4.800000000000001

Covariance of sample: 6.000000000000001

Correlation of list1 and list2: 0.41666666666666663

Skewness of list1: 0.0

Skewness of list2: 2.9296874999999987

Skewness of list1 as sample: 1.36

Skewness of list2 as sample: 2.5999999999999996

Kurtosis of list1: 0.0839506172839506

Kurtosis of list2: 7.934570312499996

Kurtosis of list1 as sample: 1.36

Kurtosis of list2 as sample: 2.5999999999999996

**[Chapter 4: Classes and Methods](#TableOfContents)**

So far, we have dealt only with functions. Functions are convenient because they generalize some exercise given a certain type of input. In the last chapter we created a function that takes the mean value of a list of elements. It may be useful to create a function that is not owned by a class if you are in a hurry, but it is better to develop a habit of building class objects whenever you think you might want to reuse the functions that we have made. To take advantage of a function while scripting in a different file, we can import the file and instantiate a class object that owns these functions. When a function is owned by a class, we refer to this as a method. In this chapter, you will learn how to create a class with methods.

**a. Arithmetic Class**

It is useful to build a class with a collection of related objects. We will start by building a class that performs basic arthimetic operations. It will include the functions "add", "multiply", and "power". Before we make any methods, however, we must initialize the class as an object itself.

We start by building the Arithmetic class and describing its \_\_init\_\_ function. This function will be called automatically upon the creation of an instance of the class. The init function will create an object that can be called at any time.

self.targetValue is used to hold the value of interest to the function

Be sure to place the class at the top of file, just after you import any libraries that you plan to use. Copy the text below to build your first class.

1. **class** Arithmetic():
2. **def** \_\_init\_\_(self):
3. self.target\_value = 0

We can create an object that is an instance of the class. From this object, we can call self.target\_value.

2. arithmetic = Arithmetic()
3. **print**(arithmetic.target\_value)

Output:

0

Following the instance of the *Arithmetic* class with a ‘.’ enables the calling of objects owned by the class.

Next, let's create the "add()" method.

1. **class** Arithmetic():
2. **def** \_\_init\_\_(self):
3. self.target\_value = 0
5. **def** add(self, \*args):
6. **try**:
7. self.target\_value = sum([arg **for** arg **in** args])
8. **return** self.target\_value
9. **except**:
10. **print**("Pass int or float to add()")

To account for inputs that cannot be processed, the method begins with *try*.This will return an error message in cases where integers or floats may not be passed to the method.

The *add()* method passes two arguments: *self* and *\*args*. Self is always implicitly passed to a method, so you will only pass one arguments that will be interpreted as part *\*args*. The *\*args* command accepts an undefined number of arguments. It is returned within the function as a tuple that includes the values passed to add. Using a for-loop, each of the values can be called individually from the tuple. We create a list from the arguments passed using a generator function, summing the list.

Pass values to the *add* method as noted below

1. arithmetic = Arithmetic()
2. **print**(arithmetic.target\_value)
3. **print**(arithmetic.add(2,3,4))

Output:

0

9

We will add two more functions to our class: the multiply and power functions. As with the addition class, we will create a multiply class that multiplies an unspecified number of arguments.

1. #arithmetic.py
2. **from** functools **import** reduce
3. **from** operator **import** mul
5. **class** Arithmetic():
6. **def** \_\_init\_\_(self):
7. self.target\_value = 0
9. **def** add(self, \*args):
10. **try**:
11. self.target\_value = sum([arg **for** arg **in** args])
12. **return** self.target\_value
13. **except**:
14. **print**("Pass int or float to add()")
16. **def** multiply(self, \*args):
17. self.target\_value = 1
18. **try**:
19. self.target\_value = reduce(mul, args)
20. **return** self.target\_value
21. **except**:
22. **print**("Pass int or float to multiply()")

To multiply the arguments, we first import *reduce* from the *functools* library and *mul* from the *operator* library. *reduce* simplifies multiple values to a single value through a specified means. We use mul to specify that we want the product of the terms.

1. arithmetic = Arithmetic()
2. **print**(arithmetic.target\_value)
3. **print**(arithmetic.add(2,3,4))
4. **print**(arithmetic.multiply(2,3,4))

Output:

0

9

24

The last method we will create is the exponent function. This one is straight-forward. Pass a base and an exponent to *power()* to yield the result a value, , where .

1. #arithmetic.py
2. **from** functools **import** reduce
3. **from** operator **import** mul
5. **class** Arithmetic():
6. **def** \_\_init\_\_(self):
7. self.target\_value = 0
9. **def** add(self, \*args):
10. **try**:
11. self.target\_value = sum([arg **for** arg **in** args])
12. **return** self.target\_value
13. **except**:
14. **print**("Pass int or float to add()")
16. **def** multiply(self, \*args):
17. self.target\_value = 1
18. **try**:
19. self.target\_value = reduce(mul, args)
20. **return** self.target\_value
21. **except**:
22. **print**("Pass int or float to multiply()")
24. **def** power(self, base, exponent):
25. self.target\_value = base \*\* exponent
26. **return** self.target\_value

1. arithmetic = Arithmetic()
2. **print**(arithmetic.target\_value)
3. **print**(arithmetic.add(2,3,4))
4. **print**(arithmetic.multiply(2,3,4))
5. **print**(arithmetic.power(2,3))

Output:

0

9

24

8

**b. Stats Class**

Now that you are comfortable with classes, we can build a Stats() class. This

will integrate of the core stats functions that we built in the last chapter.

We will be making use of this function when we build a program to run ordinary

least squares regression, so make sure that this is well ordered.

Since we have already built the stats functions, I have included the script

below and run each function once to check that the class is in working order.

Note that everytime a function owned by the Stats() class is called, the program

must first call "self". This calls the objects itself. We follow self with

".function-name". For example, the mean function must call the total function.

It does so with the command "self.total(listObj)".

1. **class** Stats():
2. **def** \_\_init\_\_(self):
3. self.targetValue = 0
5. **def** total(self, listObj):
6. total = 0
7. **for** i **in** range(len(listObj)):
8. total += listObj[i]
9. **return** total
11. **def** mean(self, listObj):
12. mean = self.total(listObj) / len(listObj)
13. **return** mean
15. **def** median(self, listObj):
16. median = 0
17. **if** len(listObj) % 2 != 0:
18. index = int((len(listObj) - 1) / 2)
19. median = float(listObj[index])
20. **else**:
21. index1 = int((len(listObj)) / 2)
22. index2 = index1 - 1
23. median = (listObj[index1] + listObj[index2]) / 2
24. **return** median
26. **def** mode(self, listObj):
27. modeValues = []
28. maxCount = 0
29. counterDict = {}
30. **for** value **in** listObj:
31. counterDict[value] = 0
33. **for** value **in** listObj:
34. counterDict[value] += 1
36. **for** key **in** counterDict:
37. **if** counterDict[key] > maxCount:
38. maxCount = counterDict[key]
39. **for** key **in** counterDict:
40. **if** counterDict[key] == maxCount:
41. modeValues.append(key)
42. **return** sorted(modeValues)
44. **def** variance(self, listObj, sample = False):
45. meanval = self.mean(listObj)
46. var = 0
47. n = len(listObj)
48. **for** x **in** listObj:
49. var += (x - meanval) \*\* 2
50. **if** sample == False:
51. var = var / n
52. **if** sample:
53. var = var / (n - 1)
54. **return** var

57. **def** stdev(self, listObj, sample = False):
58. **if** sample == False:
59. stdev = self.variance(listObj) \*\* (1/2)
60. **if** sample:
61. stdev = self.variance(listObj, True) \*\* (1/2)
62. **return** stdev
64. **def** covariance(self, listObj1, listObj2, sample = False):
65. mean1 = self.mean(listObj1)
66. mean2 = self.mean(listObj2)
67. cov = 0
68. lengthList1 = len(listObj1)
69. **if** lengthList1 == len(listObj2):
71. **for** i **in** range(lengthList1):
72. cov += (listObj1[i] - mean1) \* (listObj2[i] - mean2)
74. **if** sample == False:
75. cov = cov / (lengthList1)
76. **if** sample == True:
77. cov = cov / (lengthList1 - 1)
79. **return** cov
80. **else**:
81. **print**("List lengths are not equal")
82. quit()
84. **def** correlation(self, listObj1, listObj2, sample = False):
86. # Same result occurs whether you use sample or mode population values
87. **if** sample == False:
88. cov = self.covariance(listObj1, listObj2)
89. stdev1 = self.stdev(listObj1)
90. stdev2 = self.stdev(listObj2)
91. **if** sample:
92. cov = self.covariance(listObj1, listObj2, True)
93. stdev1 = self.stdev(listObj1, True)
94. stdev2 = self.stdev(listObj2, True)
95. corr = cov / (stdev1 \* stdev2)
96. **return** corr
98. **def** skew(self, listObj, sample = False):
99. mean = self.mean(listObj)
100. skew = 0
101. n = len(listObj)
102. **for** x **in** listObj:
103. skew += (x - mean) \*\* 3
104. **if** sample == False:
105. skew = skew / n
106. stdev = self.stdev(listObj)
107. **if** sample:
108. skew = skew / (n - 1)
109. stdev = self.stdev(listObj, True)
111. skew = skew / (stdev \*\* 3)
112. **return** skew

115. **def** kurt(self, listObj, sample = False):
116. mean = self.mean(listObj)
117. kurt = 0
118. n = len(listObj)
119. **for** x **in** listObj:
120. kurt += (x - mean) \*\* 4
121. **if** sample == False:
122. stdev = self.stdev(listObj)
123. kurt = kurt / n
124. **if** sample:
125. stdev = self.stdev(listObj, True)
126. kurt = kurt / (n - 1)
128. kurt = kurt / (stdev \*\* 4)
129. **return** kurt

We create an instance of stats as follows

1. stats = Stats()

We can use this instance to call functions owned by it.

1. list1 = [1,4,7,33,5,4,22,55,4,55,4,32]
2. list2 = [4,8,22,1,9,43,3,2,1,99,3,10]
4. stats = Stats()
5. total1 = stats.total(list1)
6. total2 = stats.total(list2)
7. mean1 = stats.mean(list1)
8. mean2 = stats.mean(list2)
9. mode1 = stats.mode(list1)
10. mode2 = stats.mode(list2)
11. median1 = stats.median(list1)
12. median2 = stats.median(list2)
13. variance1 = stats.variance(list1)
14. variance2 = stats.variance(list2)
15. standardDeviation1 = stats.stdev(list1)
16. standardDeviation2 = stats.stdev(list2)
17. covariancePop = stats.covariance(list1, list2)
18. covarianceSample = stats.covariance(list1, list2, True)
19. correlationPop = stats.correlation(list1, list2)
20. correlationSample = stats.correlation(list1, list2, True)
21. skewnessPop1 = stats.skew(list1)
22. skewnessPop2 = stats.skew(list2)
23. skewnessSample1 = stats.skew(list1, True)
24. skewnessSample2 = stats.skew(list2, True)
25. kurtosisPop1 = stats.kurt(list1)
26. kurtosisPop2 = stats.kurt(list1)
27. kurtosisSample1 = stats.kurt(list1, True)
28. kurtosisSample2 = stats.kurt(list1, True)

Output:

Stats Instance: <stats.Stats object at 0x00000155074CBBA8>

Total1: 226

Total2: 205

Mean1: 18.833333333333332

Mean2 17.083333333333332

Mode1: [4]

Mode2: [1, 3]

Median1: 13.0

Median2: 23.0

Variance1: 377.47222222222223

Variance2: 743.0763888888888

Standard Deviation1: 19.428644374279493

Standard Deviation2: 27.259427523132043

Covariance (Population): 211.34722222222229

Covariance (Sample): 230.56060606060612

Correlation (Population): 0.3990591877942193

Correlation (Sample): 0.39905918779421923

SkewnessPop1 (Population) 0.8720032076106592

SkewnessPop2 (Population) 2.2462815894195556

SkewnessSample1 (Sample) 0.834879509016923

SkewnessSample2 (Sample) 2.1506508853642745

Kurtosis1 (Population) 0.8720032076106592

Kurtosis2 (Population) 2.2462815894195556

Kurtosis1 (Sample) 0.8720032076106592

Kurtosis2 (Sample) 2.2462815894195556

**[5. Working with n](#TableOfContents)*[umpy](#TableOfContents)* [and](#TableOfContents) *[pandas](#TableOfContents)***

In addition to lists and dictionaries, there exists data structures from the *numpy* and *pandas* libraries that you should be aware of. The *numpy* library uses structures that are more efficient than lists. Pandas, which uses *numpy*, helps to organize data in tables.

**a. *numpy***

**i. Arrays**

Lists in Python do not need to be cast a particular type of object such as float, string, or object. This makes programming in Python relatively simple. Lists in Python are dynamic, meaning that they are not fixed in size. This comes with the drawback that execution time is slower than it otherwise could be.

Although it is a library in Python, numpy functions are programmed in C++. The *numpy* library solves this problem by creating arrays that contain data types that are values. These include integers and a variety of floats. Both of these aspects contribute to a substantial increase in efficiency for computation performed using numby arrays instead of dynamic lists.

There are a number of ways to create a *numpy* array. The easiest is simply to convert a python list using the command *array*.

1. #numpyArray.py
2. **import** numpy as np
4. array = np.array([1, 2, 3, 4, 5])
5. **print**(array)

Output:

[1 2 3 4 5]

We have created an array of 32-bit integers. We can check the data type using the command *dtype* after the array object

1. #numpyArray.py
2. **import** numpy as np
4. array = np.array([1, 2, 3, 4, 5])
5. **print**(array)
6. **print**(array.dtype)

Output:

[1 2 3 4 5]

int32

If we include a decimal point with any single value, the type will change to a float.

1. #numpyArrayFloat.py
2. **import** numpy as np
4. array = np.array([1, 2., 3, 4, 5])
5. **print**(array)
6. **print**(array.dtype)

Output:

[ 1. 2. 3. 4. 5.]

float64

Numpy also includes command that allow you to quickly make arrays of 1) a range of numbers, 2) zeros, and 3) ones. The *arange()* takes the input (start, finish, interval). If only one number is entered, it assumes that the start of the range is 0. If two numbers are entered, the first number represents the start of the range. The highest values input into the array will be one less than the second number. Below we create an array with range [0,100]

1. #numpyInstantArrays.py
2. **import** numpy as np
4. rangeArray = np.arange(0,101)
5. **print**(rangeArray)

Output:

[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53

54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71

72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89

90 91 92 93 94 95 96 97 98 99 100]

We can create another array using the same range, but this time let’s count by two.

1. #numpyInstantArrays.py
2. **import** numpy as np
4. rangeArray = np.arange(0,101)
5. rangeArray2 = np.arange(0,101, 2)
7. **print**(rangeArray)
8. **print**(rangeArray2)

Output:

[ 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34

36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70

72 74 76 78 80 82 84 86 88 90 92 94 96 98 100]

So far, we have only worked with lists of one dimension. Lists in Python may have multiple dimension. Below we create a two dimensional list:

1. #twoDimensionalListAndNumpyArray.py
2. **import** numpy as np
4. twoDimList = [[1,2,3,4],[2,3,4,5]]
5. **print**(twoDimList)

Output:

[[1, 2, 3, 4], [2, 3, 4, 5]]

This is a list of lists. This can be read as a list with two elements. Those two elements are both list with four elements that are integers. We can print each of this lists in *twoDimList* by using a for loop:

1. #twoDimensionalListAndNumpyArray.py
2. **import** numpy as np
4. twoDimList = [[1,2,3,4],[2,3,4,5]]
5. **print**(twoDimList)
7. **for** i **in** range(len(twoDimList)):
8. **print**(twoDimList[i])

Output:

[[1, 2, 3, 4], [2, 3, 4, 5]]

[1, 2, 3, 4]

[2, 3, 4, 5]

The elements from each of these lists can be called individually by using to index values: the first value calls the list in twoDimList, the second value calls an element form that list. We call each element by using an additional for-loop:

1. #twoDimensionalListAndNumpyArray.py
2. **import** numpy as np
4. twoDimList = [[1,2,3,4],[2,3,4,5]]
5. **print**(twoDimList)
7. **for** i **in** range(len(twoDimList)):
8. **print**(twoDimList[i])
9. **for** j **in** range(len(twoDimList[i])):
10. **print**(twoDimList[i][j])

This list of integers can be passed to a function in the numpy library to create a two-dimensional array. In the same manner that we create a one-dimensional array:

twoDimArray = np.array(twoDimList)

We can print this function and its elements using the same structure that we used for *twoDimList*.

1. #twoDimensionalListAndNumpyArray.py
2. **import** numpy as np
4. twoDimList = [[1,2,3,4],[2,3,4,5]]
5. **print**(twoDimList)
6. twoDimArray = np.array(twoDimList)
7. **print**(twoDimArray)
9. **for** i **in** range(len(twoDimList)):
10. **print**(twoDimList[i])
11. **print**(twoDimArray[i])
12. **for** j **in** range(len(twoDimList[i])):
13. **print**(twoDimList[i][j])
14. **print**(twoDimArray[i][j])

Two dimensional arrays can be useful for working keeping track of data if there is a need for a high level of efficiency. For the most part, we will work with dictionaries and dataframes, but it is useful to understand how data is structured and called using a two dimensional array.

Data in an array can be entered for each individual element. Entry of data into an array comprised of zeros helps to illustrate the point. We will enter new values into the second element of the zeroth list ([0][2]), the zeroth element of the first list ([1][0], and the second element of the second list [2][2]:

1. **import** numpy as np
3. array = np.zeros((3,3))
4. **print**(array)
6. array[0][2] = 9
7. array[1][0] = 7
8. array[2][2] = 3
9. **print**(array)

[[0. 0. 0.]

[0. 0. 0.]

[0. 0. 0.]]

[[0. 0. 9.]

[7. 0. 0.]

[0. 0. 3.]]

There are several functions that operate like the zeros function. For example, *np.ones()* return an array of ones and *np.empty()* returns an array of random values. Similar functions create an array of the same dimension as the one passed. For example, a 3X3 array passed to *np.zeros\_like()* will create a 3X3 array of zeros. We test several of these functions below:

1. #zerosOnesAndLike.py
3. listOfLists = [[1,2,3],[4,5,6],[7,8,9]]
4. arrayOfArrays = np.array(listOfLists)
5. zerosLikeArray = np.zeros\_like(listOfLists)
6. onesLikeArray = np.ones\_like(listOfLists)
7. emptyLikeArray = np.empty\_like(listOfLists)
9. **print**("listOfLists:\n", listOfLists)
10. **print**("arrayOfArrays:\n", arrayOfArrays)
11. **print**("zerosLikeArray:\n", zerosLikeArray)
12. **print**("onesLikeArray:\n", onesLikeArray)
13. **print**("emptyLikeArray:\n", emptyLikeArray)

Output:

listOfLists:

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

arrayOfArrays:

[[1 2 3]

[4 5 6]

[7 8 9]]

zerosLikeArray:

[[0 0 0]

[0 0 0]

[0 0 0]]

onesLikeArray:

[[1 1 1]

[1 1 1]

[1 1 1]]

emptyLikeArray:

[[1 2 3]

[4 5 6]

[7 8 9]]

**ii. Useful Functions and Values**

Numpy has a number of special functions and values that may be useful depending on your purpose. We will cover a few of these. For a full list see here:

<https://docs.scipy.org/doc/numpy/reference/routines.math.html>

It is helpful to be aware of this list of functions, keeping in mind that implementing them is simple if you know where to find them.

Below, we call several functions that may be useful when working with data. In particular, *np.ln()* and *np.log()* are useful for working with data that is subject to a trend. Changes in logged values approximate rates of change as compared to change in the unit of analysis.

1. #numpyLogs.py
3. vals = np.arange(0, 10001, 100)
4. ln\_vals = np.log(vals)
5. log10\_vals = np.log10(vals)
6. **print**(vals)
7. **print**(ln\_vals)
8. **print**(log10\_vals)

Output:

[ 0 100 200 300 400 500 600 700 800 900 1000 1100

1200 1300 1400 1500 1600 1700 1800 1900 2000 2100 2200 2300

2400 2500 2600 2700 2800 2900 3000 3100 3200 3300 3400 3500

3600 3700 3800 3900 4000 4100 4200 4300 4400 4500 4600 4700

4800 4900 5000 5100 5200 5300 5400 5500 5600 5700 5800 5900

6000 6100 6200 6300 6400 6500 6600 6700 6800 6900 7000 7100

7200 7300 7400 7500 7600 7700 7800 7900 8000 8100 8200 8300

8400 8500 8600 8700 8800 8900 9000 9100 9200 9300 9400 9500

9600 9700 9800 9900 10000]

[ -inf 4.60517019 5.29831737 5.70378247 5.99146455 6.2146081

6.39692966 6.55108034 6.68461173 6.80239476 6.90775528 7.00306546

7.09007684 7.17011954 7.24422752 7.31322039 7.37775891 7.43838353

7.49554194 7.54960917 7.60090246 7.64969262 7.69621264 7.7406644

7.78322402 7.82404601 7.86326672 7.90100705 7.9373747 7.97246602

8.00636757 8.03915739 8.07090609 8.10167775 8.13153071 8.16051825

8.18868912 8.2160881 8.24275635 8.26873183 8.29404964 8.31874225

8.3428398 8.3663703 8.38935982 8.41183268 8.43381158 8.45531779

8.4763712 8.49699048 8.51719319 8.53699582 8.5564139 8.5754621

8.59415423 8.61250337 8.63052188 8.64822145 8.6656132 8.68270763

8.69951475 8.71604405 8.73230457 8.74830491 8.76405327 8.77955746

8.79482493 8.80986281 8.82467789 8.83927669 8.85366543 8.86785006

8.88183631 8.89562963 8.90923528 8.9226583 8.93590353 8.94897561

8.96187901 8.97461804 8.98719682 8.99961934 9.01188943 9.02401079

9.03598698 9.04782144 9.05951748 9.0710783 9.082507 9.09380656

9.10497986 9.11602969 9.12695876 9.13776968 9.14846497 9.15904708

9.16951838 9.17988116 9.19013766 9.20029004 9.21034037]

[ -inf 2. 2.30103 2.47712125 2.60205999 2.69897

2.77815125 2.84509804 2.90308999 2.95424251 3. 3.04139269

3.07918125 3.11394335 3.14612804 3.17609126 3.20411998 3.23044892

3.25527251 3.2787536 3.30103 3.32221929 3.34242268 3.36172784

3.38021124 3.39794001 3.41497335 3.43136376 3.44715803 3.462398

3.47712125 3.49136169 3.50514998 3.51851394 3.53147892 3.54406804

3.5563025 3.56820172 3.5797836 3.59106461 3.60205999 3.61278386

3.62324929 3.63346846 3.64345268 3.65321251 3.66275783 3.67209786

3.68124124 3.69019608 3.69897 3.70757018 3.71600334 3.72427587

3.73239376 3.74036269 3.74818803 3.75587486 3.76342799 3.77085201

3.77815125 3.78532984 3.79239169 3.79934055 3.80617997 3.81291336

3.81954394 3.8260748 3.83250891 3.83884909 3.84509804 3.85125835

3.8573325 3.86332286 3.86923172 3.87506126 3.88081359 3.88649073

3.8920946 3.89762709 3.90308999 3.90848502 3.91381385 3.91907809

3.92427929 3.92941893 3.93449845 3.93951925 3.94448267 3.94939001

3.95424251 3.95904139 3.96378783 3.96848295 3.97312785 3.97772361

3.98227123 3.98677173 3.99122608 3.99563519 4. ]

Notice that changes in the logged values are much more gradual than changes in the values from the original array. This will be useful when we visualize and analyze data.

Numpy also provides values of special constants such as *π*. Below, we call a few of these values that will probably be useful to know at some point:

1. #specialValuesInNumpy.py
2. **import** numpy as np
4. pi = np.pi
5. e = np.e
6. lne = np.log(e)
7. infinity = np.inf
8. null\_val = np.nan
9. **print**("pi =", pi)
10. **print**("e =", e)
11. **print**("ln(e)=", lne)
12. **print**("infintity:", infinity)
13. **print**("null\_val:",null\_val)

Output:

pi = 3.141592653589793

e = 2.718281828459045

ln(e)= 1.0

infintity: inf

nullVal: nan

As with the list of functions, it is worthwhile to commit to memory the commands for special values that you expect to use in your work. A full list of constants can be found here:

<https://docs.scipy.org/doc/numpy/reference/constants.html>

**iii. Indexing**

The *numpy* library allows users to select a subset of values in light of a particular criteria. In order to accomplish this, we would need to nest for loops and if functions to construct an array. In the following examples, we will create similar output using commands that are native to Python and those that are created using the *numpy* library.

Suppose that you want to construct an array of even numbers. The following command would suffice:

1. #evenNumbers.py
3. even\_nums = [i **for** i **in** range(100) **if** i % 2 == 0]
4. **print**(even\_nums)

Output:

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]

We create an array of even numbers using a generator function. The use of a generator function is concise. It tells python to record every value I from 0 to 99 that has a remainder of zero when divided by 2. We can accomplish the same task using a numpy array.

1. #evenNumbers.py
2. **import** numpy as np
4. even\_nums = [i **for** i **in** range(100) **if** i % 2 == 0]
5. **print**(even\_nums)
7. even\_nums = np.arange(100)
8. even\_nums = even\_nums[even\_nums % 2 == 0]
9. **print**(even\_nums)

Ouput:

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]

[ 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46

48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94

96 98]

Passing passing a constraint to the index returns an array whose values satisfy the constraint. In this case, only values divisible by 2 are returned.

You probably won’t be looking for even numbers in your data. You may want to only include data that fits some minimum or maximum criteria. In the next example, we will produce random numbers and include only those numbers whose values satisfy a minimum constraint. The

1. #booleanIndexing.py
2. **import** random
4. rand\_list = [random.random() \* 10 **for** i **in** range(7)]
5. **print**("rand\_list values:", rand\_list)
6. rand\_list = [i **for** i **in** rand\_list **if** i > 3]
7. **print**("rand\_list values > 3:", rand\_list)

rand\_list values: [9.014400035181964, 7.113405883920313, 1.1429171214071765, 8.586953492641646, 9.709404949038433, 7.949358347593969, 2.5285466560357364]

rand\_list values > 3: [9.014400035181964, 7.113405883920313, 8.586953492641646, 9.709404949038433, 7.949358347593969]

We could have produced the same output by adding the if statement to the end of the for loop in the previous generator function. We would not, however, been able to see the full range of values produced.

We follow the same procedure as before to produce a numpy array whose values satisfy the constraint of being greater than 3.

1. #booleanIndexing.py
2. **import** random
3. **import** numpy as np
5. rand\_list = [random.random() \* 10 **for** i **in** range(7)]
6. **print**("rand\_list values:", rand\_list)
7. rand\_list = [i **for** i **in** rand\_list **if** i > 3]
8. **print**("rand\_list values > 3:", rand\_list)
10. rand\_array = np.random.randn(7) \* 10
11. **print**("rand\_array:", rand\_array)
12. **print**("rand\_array > 3:", rand\_array[rand\_array > 3])

rand\_list values: [9.014400035181964, 7.113405883920313, 1.1429171214071765, 8.586953492641646, 9.709404949038433, 7.949358347593969, 2.5285466560357364]

rand\_list values > 3: [9.014400035181964, 7.113405883920313, 8.586953492641646, 9.709404949038433, 7.949358347593969]

rand\_array: [ 1.88115064 -20.18344745 8.43029667 14.26415369 8.70881611

15.59822709 15.67472772]

rand\_array > 3: [ 8.43029667 14.26415369 8.70881611 15.59822709 15.67472772]

**b. *pandas***

**i. Dictionaries**

Numpy enables efficient computation, however, organizing data is a challenge if one is using only structures from *numpy*. One way of organizing data is to use a dictionary. While dictionaries are capable of holding any kind of object, they are especially well suited for managing data. Instead of creating multi-dimensional arrays to hold data, a dictionary allows us to enter a single vector of data and to pair this vector with a name. The name is the *key* that calls the vector.

The *pandas* library operates like a dictionary with special structure and functions. To show this, we will create a dictionary with data from numpy arrays.

1. #dataDictpy
2. **import** numpy as np

5. dataDict = {"1 to 10":np.arange(10),
6. "ones":np.ones(10),
7. "zeros":np.zeros(10)}
8. **print**(dataDict)

Output:

{'1 to 10': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 'ones': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), 'zeros': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])}

Printing the dictionary shows the data attached to the key. This does not look very well organized. Let’s see what happens if we call each key indvidiually and print the output.

1. #dataDictpy
2. **import** numpy as np
4. dataDict = {"1 to 10":np.arange(10),
5. "ones":np.ones(10),
6. "zeros":np.zeros(10)}
7. **print**(dataDict)
8. **print**([dataDict[key] **for** key **in** dataDict])

Output:

{'1 to 10': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 'ones': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), 'zeros': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])}

[array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])]

Ths result did not improve the readability of the output. And the keys disappeared!

Let’s use the print function to clean up the presentation. We will use a generator function to print each key and the array that it calls in *dataDict*.

1. #dataDictpy
2. **import** numpy as np
4. dataDict = {"1 to 100":np.arange(10),
5. "ones":np.ones(10),
6. "zeros":np.zeros(10)}
7. **print**(dataDict)
8. **print**([dataDict[key] **for** key **in** dataDict])
9. **print**([key + " " + str(dataDict[key]) **for** key **in** dataDict])

Output:

{'1 to 10': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 'ones': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), 'zeros': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])}

[array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])]

['1 to 10 [0 1 2 3 4 5 6 7 8 9]', 'ones [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]', 'zeros [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]']

We can also call only a portion of each array by calling a slice of it. Alter the string passed to *print()* by adding a slice, ‘[5:10]’.

1. #dataDictpy
2. **import** numpy as np
4. dataDict = {"1 to 10":np.arange(10),
5. "ones":np.ones(10),
6. "zeros":np.zeros(10)}
7. **print**(dataDict)
8. **print**([dataDict[key] **for** key **in** dataDict])
9. **print**([key + " " + str(dataDict[key])  **for** key **in** dataDict])
10. **print**([key + " " + str(dataDict[key][5:10])  **for** key **in** dataDict])

Output:

{'1 to 10': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), 'ones': array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), 'zeros': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])}

[array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]), array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])]

['1 to 10 [0 1 2 3 4 5 6 7 8 9]', 'ones [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]', 'zeros [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]']

['1 to 10 [5 6 7 8 9]', 'ones [1. 1. 1. 1. 1.]', 'zeros [0. 0. 0. 0. 0.]']

ii. DataFrame

Dictionaries are useful for organizing data, however, presentation of data could be improved. The *pandas* library retains all of the structural advantages of a dictionary. It also improves organization by simplifying processes required for calling and storing data, as well as improving visualization of the data structure.

The *pandas* libraries includes *Series()* and *DataFrame()*, which call data structures. *Series()* can store a single column of data with and index and a column name. *DataFrame()* can hold multiple vectors of data, each called by a key that serves to name the vector of data.

You can pass a dictionary with a single vector to either a *Series()* or a *DataFrame()*. Typically, *DataFrame()* serve the same purpose as *Series()*. According to the *pandas* API, “Can be thought of as a dict-like container for Series objects.” A DataFrame adds the benefit of a key that is used to call the series.

1. #pandasSeriesVsDataFrame.py
2. **import** numpy as np
3. **import** pandas as pd
5. dataDict = {"range":np.arange(10)}
6. dataSeries = pd.Series(dataDict)
7. **print**(dataSeries)

Output:

range [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

dtype: object

We could pass only an array, rather than a dictionary holding an array, to *Series()*. The result includes index numbers (left column), but lacks a key.

series = pd.Series(np.arange(10))

**print**(series)

Output:

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

dtype: int32

If we call the key, “range”, for *dataSeries*, then the array will be called, but it will not be indexed:

1. #pandasSeriesVsDataFrame.py
2. **import** numpy as np
3. **import** pandas as pd
5. dataDict = {"range":np.arange(10)}
6. dataSeries = pd.Series(dataDict)
7. **print**(dataSeries)
8. **print**(dataSeries["range"])

Output:

range [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

dtype: object

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

It is usually more convenient to work with DataFrame should be used instead of Series. A DataFrame holds one or miultiples Series each associated with a key. Whereas, Series treats the key of a Dictionary as an indexer, DataFrame will treat the key as a column name.

To create a DataFrame with the dataDict, pass the dictionary to *DataFrame()*,then print the resultant DataFrame:

1. #pandasSeriesVsDataFrame.py
2. **import** numpy as np
3. **import** pandas as pd
5. dataDict = {"range":np.arange(10)}
6. dataSeries = pd.Series(dataDict)
7. **print**(dataSeries)
8. **print**(dataSeries["range"])
10. dataDF=pd.DataFrame(dataDict)
11. **print**(dataDF)

Output:

range [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

dtype: object

range

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

Values in a DataFrame are typically called by key and index. Calling *dataDF[“range”]* will return the Series that identified by that key. The data will not be preceded by the key, however the Series will include it at the bottom:

1. #pandasSeriesVsDataFrame.py
2. **import** numpy as np
3. **import** pandas as pd
5. dataDict = {"range":np.arange(10)}
6. dataSeries = pd.Series(dataDict)
7. **print**(dataSeries)
8. **print**(dataSeries["range"])
10. dataDF=pd.DataFrame(dataDict)
11. **print**(dataDF)
12. **print**(dataDF["range"])

Output:

range [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

dtype: object

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

range

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

Name: range, dtype: int32

As with the dictionary, we can call a slice of the DataFrame. This can be done one of two ways. Either call the desired column or columns and follow with the index values.

1. #pandasSeriesVsDataFrame.py
2. **import** numpy as np
3. **import** pandas as pd
5. dataDict = {"range":np.arange(10)}
6. dataSeries = pd.Series(dataDict)
7. **print**(dataSeries)
8. **print**(dataSeries["range"])
10. dataDF=pd.DataFrame(dataDict)
11. **print**(dataDF)
12. **print**(dataDF["range"])
13. **print**(dataDF["range"][5:9])

Output:

range [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

dtype: object

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

range

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

Name: range, dtype: int32

5 5

6 6

7 7

8 8

Name: range, dtype: int32

You may also use the loc command, which allows you to call the index without calling a particular column.

dataDF.loc[5:9]

This would return:

Output:

range

5 5

6 6

7 7

8 8

9 9

ii. Using a Dictionary to Specify a DataFrame’s Index

When you save arrays in a dictionary, they do not include any particular index. Instead, *pandas* creates a general index from 0 to one less than the length of the array. If you do not need a particular index, this can work.

It is often useful to create a dictionary and identify each value of interest individually. If you are a macroeconomist, you may, for example, collect data for “GDP”, “Real GDP”, the “Price Level”, and the quantity of “Money” (each key is in quotes). Below, we prepare a dictionary with values for years from 1990 to 2018.

1. #dictWithIndexAndKey.py
2. **import** pandas as pd
3. **import** numpy as np
5. macroDict = {"GDP":{},
6. "Real GDP":{},
7. "Price Level":{},
8. "Money":{}}
10. **for** key **in** macroDict:
11. **for** i **in** range(1990,2018):
12. macroDict[key][i] = np.random.random()\* 10000
14. **print**(macroDict)

Output:

{'GDP': {1990: 7065.235949212676, 1991: 7753.848598497283, 1992: 6885.782543995146, 1993: 1316.9724301227836, 1994: 814.1427090102737, 1995: 8643.892233223702, 1996: 2646.147304271733, 1997: 4254.4051730276915, 1998: 8398.410691381048, 1999: 5424.022680978063, 2000: 6100.619668460457, 2001: 8685.476568332842, 2002: 9550.89893581937, 2003: 8197.235143503838, 2004: 4632.912503201125, 2005: 3118.039926614239, 2006: 7769.660442320114, 2007: 1169.9074226990124, 2008: 8950.737558230472, 2009: 8609.876524318706, 2010: 9750.79706797999, 2011: 8503.835864002898, 2012: 5292.876012614783, 2013: 4616.872630108899, 2014: 6660.53339421941, 2015: 9498.064409816518, 2016: 7204.253912744478, 2017: 1378.2725551436038}, 'Real GDP': {1990: 3420.4638043905943, 1991: 6539.4868656442195, 1992: 9957.672641727508, 1993: 5211.858930056559, 1994: 2581.7244981653585, 1995: 3648.0380804374213, 1996: 4126.009998067315, 1997: 8594.43956683996, 1998: 5654.009403490598, 1999: 6791.412425352907, 2000: 4604.172908731939, 2001: 5883.814950356392, 2002: 1296.7500195504022, 2003: 2547.7243542715855, 2004: 5637.393206610598, 2005: 6331.485814297805, 2006: 6269.7359590667775, 2007: 6909.798382184205, 2008: 4357.699780885533, 2009: 9793.913250441352, 2010: 1518.6151829447604, 2011: 7716.490857211774, 2012: 2632.0289267425246, 2013: 9563.105342134962, 2014: 3985.979765410654, 2015: 7562.766808314471, 2016: 2581.501721603301, 2017: 6707.991339879588}, 'Price Level': {1990: 3116.2402518221857, 1991: 8512.711404978903, 1992: 3178.6177088021427, 1993: 6168.8875436238595, 1994: 4690.915223803815, 1995: 212.87725096827637, 1996: 4933.897863021942, 1997: 5098.477801321568, 1998: 6136.3784050380345, 1999: 147.2889689847434, 2000: 8251.196763330581, 2001: 1802.8554402006946, 2002: 9458.575032895294, 2003: 8530.312216538547, 2004: 4761.270119933782, 2005: 6445.602846168615, 2006: 7505.838923158045, 2007: 8762.168524250297, 2008: 1802.946811050249, 2009: 9750.850561938576, 2010: 6297.159004892377, 2011: 7463.662623070008, 2012: 8632.537350326416, 2013: 9742.927892741734, 2014: 3556.5458995871204, 2015: 2769.4962620808174, 2016: 4942.279126417068, 2017: 1361.5623413378753}, 'Money': {1990: 1630.038542558979, 1991: 1912.3101761315552, 1992: 1459.7875862192511, 1993: 4121.955026727403, 1994: 1808.338481548305, 1995: 4363.81935673095, 1996: 7859.279793054758, 1997: 762.6195296106152, 1998: 8277.22205517031, 1999: 6032.008134912075, 2000: 6597.799218970702, 2001: 752.7373461822496, 2002: 6036.086715910559, 2003: 3336.8597174653237, 2004: 3620.3042173943077, 2005: 7195.7744867704905, 2006: 8173.813300305426, 2007: 3002.7576831987635, 2008: 2057.330472543402, 2009: 7354.136354553914, 2010: 6425.797000222333, 2011: 6593.483976135392, 2012: 5290.203479058772, 2013: 3445.011720382095, 2014: 6705.679727781765, 2015: 6847.3641039278145, 2016: 1086.6929128920533, 2017: 8580.965514182397}}

Simply passing this to *DataFrame()* will create a DataFrame that uses the years as indexers:

1. #dictWithIndexAndKey.py
2. **import** pandas as pd
3. **import** numpy as np
5. macroDict = {"GDP":{},
6. "Real GDP":{},
7. "Price Level":{},
8. "Money":{}}
10. **for** key **in** macroDict:
11. **for** i **in** range(1990,2018):
12. macroDict[key][i] = np.random.random()\* 10000
14. **print**(macroDict)
15. macroDF = pd.DataFrame(macroDict)
16. **print**(macroDF)

{'GDP': {1990: 7065.235949212676, 1991: 7753.848598497283, 1992: 6885.782543995146, 1993: 1316.9724301227836, 1994: 814.1427090102737, 1995: 8643.892233223702, 1996: 2646.147304271733, 1997: 4254.4051730276915, 1998: 8398.410691381048, 1999: 5424.022680978063, 2000: 6100.619668460457, 2001: 8685.476568332842, 2002: 9550.89893581937, 2003: 8197.235143503838, 2004: 4632.912503201125, 2005: 3118.039926614239, 2006: 7769.660442320114, 2007: 1169.9074226990124, 2008: 8950.737558230472, 2009: 8609.876524318706, 2010: 9750.79706797999, 2011: 8503.835864002898, 2012: 5292.876012614783, 2013: 4616.872630108899, 2014: 6660.53339421941, 2015: 9498.064409816518, 2016: 7204.253912744478, 2017: 1378.2725551436038}, 'Real GDP': {1990: 3420.4638043905943, 1991: 6539.4868656442195, 1992: 9957.672641727508, 1993: 5211.858930056559, 1994: 2581.7244981653585, 1995: 3648.0380804374213, 1996: 4126.009998067315, 1997: 8594.43956683996, 1998: 5654.009403490598, 1999: 6791.412425352907, 2000: 4604.172908731939, 2001: 5883.814950356392, 2002: 1296.7500195504022, 2003: 2547.7243542715855, 2004: 5637.393206610598, 2005: 6331.485814297805, 2006: 6269.7359590667775, 2007: 6909.798382184205, 2008: 4357.699780885533, 2009: 9793.913250441352, 2010: 1518.6151829447604, 2011: 7716.490857211774, 2012: 2632.0289267425246, 2013: 9563.105342134962, 2014: 3985.979765410654, 2015: 7562.766808314471, 2016: 2581.501721603301, 2017: 6707.991339879588}, 'Price Level': {1990: 3116.2402518221857, 1991: 8512.711404978903, 1992: 3178.6177088021427, 1993: 6168.8875436238595, 1994: 4690.915223803815, 1995: 212.87725096827637, 1996: 4933.897863021942, 1997: 5098.477801321568, 1998: 6136.3784050380345, 1999: 147.2889689847434, 2000: 8251.196763330581, 2001: 1802.8554402006946, 2002: 9458.575032895294, 2003: 8530.312216538547, 2004: 4761.270119933782, 2005: 6445.602846168615, 2006: 7505.838923158045, 2007: 8762.168524250297, 2008: 1802.946811050249, 2009: 9750.850561938576, 2010: 6297.159004892377, 2011: 7463.662623070008, 2012: 8632.537350326416, 2013: 9742.927892741734, 2014: 3556.5458995871204, 2015: 2769.4962620808174, 2016: 4942.279126417068, 2017: 1361.5623413378753}, 'Money': {1990: 1630.038542558979, 1991: 1912.3101761315552, 1992: 1459.7875862192511, 1993: 4121.955026727403, 1994: 1808.338481548305, 1995: 4363.81935673095, 1996: 7859.279793054758, 1997: 762.6195296106152, 1998: 8277.22205517031, 1999: 6032.008134912075, 2000: 6597.799218970702, 2001: 752.7373461822496, 2002: 6036.086715910559, 2003: 3336.8597174653237, 2004: 3620.3042173943077, 2005: 7195.7744867704905, 2006: 8173.813300305426, 2007: 3002.7576831987635, 2008: 2057.330472543402, 2009: 7354.136354553914, 2010: 6425.797000222333, 2011: 6593.483976135392, 2012: 5290.203479058772, 2013: 3445.011720382095, 2014: 6705.679727781765, 2015: 6847.3641039278145, 2016: 1086.6929128920533, 2017: 8580.965514182397}}

GDP Real GDP Price Level Money

1990 7065.235949 3420.463804 3116.240252 1630.038543

1991 7753.848598 6539.486866 8512.711405 1912.310176

1992 6885.782544 9957.672642 3178.617709 1459.787586

1993 1316.972430 5211.858930 6168.887544 4121.955027

1994 814.142709 2581.724498 4690.915224 1808.338482

1995 8643.892233 3648.038080 212.877251 4363.819357

1996 2646.147304 4126.009998 4933.897863 7859.279793

1997 4254.405173 8594.439567 5098.477801 762.619530

1998 8398.410691 5654.009403 6136.378405 8277.222055

1999 5424.022681 6791.412425 147.288969 6032.008135

2000 6100.619668 4604.172909 8251.196763 6597.799219

2001 8685.476568 5883.814950 1802.855440 752.737346

2002 9550.898936 1296.750020 9458.575033 6036.086716

2003 8197.235144 2547.724354 8530.312217 3336.859717

2004 4632.912503 5637.393207 4761.270120 3620.304217

2005 3118.039927 6331.485814 6445.602846 7195.774487

2006 7769.660442 6269.735959 7505.838923 8173.813300

2007 1169.907423 6909.798382 8762.168524 3002.757683

2008 8950.737558 4357.699781 1802.946811 2057.330473

2009 8609.876524 9793.913250 9750.850562 7354.136355

2010 9750.797068 1518.615183 6297.159005 6425.797000

2011 8503.835864 7716.490857 7463.662623 6593.483976

2012 5292.876013 2632.028927 8632.537350 5290.203479

2013 4616.872630 9563.105342 9742.927893 3445.011720

2014 6660.533394 3985.979765 3556.545900 6705.679728

2015 9498.064410 7562.766808 2769.496262 6847.364104

2016 7204.253913 2581.501722 4942.279126 1086.692913

2017 1378.272555 6707.991340 1361.562341 8580.965514

Once the DataFrame has been created, it is easy to create new values using data from the existing DataFrame. For example, the velocity of money is calculated as the quantity of money divided by GDP (these are random values, but let’s assume that they are real). We create a new column in the DataFrame:

1. #dictWithIndexAndKey.py
2. **import** pandas as pd
3. **import** numpy as np
5. macroDict = {"GDP":{},
6. "Real GDP":{},
7. "Price Level":{},
8. "Money":{}}
10. **for** key **in** macroDict:
11. **for** i **in** range(1990,2018):
12. macroDict[key][i] = np.random.random()\* 10000
14. **print**(macroDict)
15. macroDF = pd.DataFrame(macroDict)
16. macroDF["Velocity"] = macroDF["Money"] / macroDF["GDP"]
17. **print**(macroDF)

The new DataFrame includes Velocity:

GDP Real GDP Price Level Money Velocity

1990 538.921576 7731.852384 3100.275909 4453.274622 8.263307

1991 9450.812533 9450.484348 6552.569636 8822.329318 0.933500

1992 8461.583635 2632.409053 3634.843823 4943.319522 0.584207

1993 4633.507746 2302.102808 5118.575844 6756.752222 1.458237

1994 1334.909502 2888.585304 4977.566681 7855.608141 5.884750

1995 6854.384002 5828.561150 5065.441814 5897.617861 0.860415

1996 8169.470853 7419.981271 8315.466394 449.667221 0.055042

1997 1878.231251 7942.913208 1075.189751 3544.755855 1.887284

1998 1316.040006 6439.132166 8171.172825 4718.896679 3.585679

1999 4786.349987 3578.504275 5245.666133 3856.848281 0.805802

2000 6298.900451 8444.816608 6440.230154 3113.855683 0.494349

2001 591.004162 7938.029297 8931.494938 6731.069850 11.389209

2002 7202.512199 2217.101940 5680.870883 5076.590833 0.704836

2003 6399.212706 4438.352470 4978.807857 56.934659 0.008897

2004 4230.555678 912.714189 2681.757413 8620.253739 2.037617

2005 5928.070454 7782.249044 5962.687027 4337.827021 0.731743

2006 5389.912437 7556.332809 1461.381196 2037.630971 0.378045

2007 7223.273905 4816.875741 3991.442767 4877.931721 0.675308

2008 705.212269 6292.928205 2844.025609 9292.176780 13.176425

2009 2239.559526 3342.260721 943.964357 2542.107563 1.135093

2010 2624.147277 832.212016 9989.163215 3781.082112 1.440880

2011 4070.641566 4071.502575 5980.443619 2561.274808 0.629207

2012 4996.259244 4212.263595 33.195670 4640.362309 0.928767

2013 1740.823044 2455.661274 4999.772285 2400.569819 1.378986

2014 4324.232518 5998.614696 3261.684428 2496.436207 0.577313

2015 5036.667324 7450.661091 7440.170977 9601.267667 1.906274

2016 4940.694223 5576.597257 8575.469521 5588.613084 1.131139

2017 3517.237930 1326.783888 1029.134845 2394.417535 0.680766

As with the previous example, we can call a slice of this DataFrame using *.loc()*. The following command calls data from 1995 to 2000:

macroDF.loc[1995:2000]

This returns the following DataFrame:

GDP Real GDP Price Level Money

1995 437.592394 5834.413985 4877.975172 2334.989471

1996 4458.781489 8176.564593 8525.127698 5273.655699

1997 5043.752766 806.534816 3164.899053 6888.495715

1998 1327.365771 9517.712483 4569.378913 541.782450

1999 1365.355259 9325.889858 4483.494058 674.164921

2000 6130.715198 3740.517449 7559.863957 6435.100469

Now that we have familiarized ourselves with basic commands for working with *pandas*, we are ready to work with real data. In the next chapter we import and clean data. We will also scrape data using the *pandas\_datareader*.

**[6. Importing, Cleaning, and Analyzing Data](#TableOfContents)**

We live in the age of data. With a little bit of searching and some idea of what you would like to investigate, you will be able to find data online. In the following exercise we will use data from the Index of Economic Freedom. Visit their downloads page

<http://www.heritage.org/index/download>

Under the section titled *2017 Index of Economic Freedom*, click the button titled *Download Raw Data.*



Figure 2.1

Download it to the same folder in which you have saved the py files that you are using this chapter. Save the data as a csv, rather than as an excel document.

* 1. Working with Data

Once you have saved it, we will import the data using *pandas* library. We use *DataFrame()* frame function to build a data frame. Using *from\_csv* we are able to import the csv as a data frame. This automatically imports the data using the elements that comprise the header as keys. The data frame works like a dictionary, but is structured like a spreadsheet when printed. The pandas library also has a number of functions that allows you to manipulate data contained in a data frame or series. We also import stats, the *stats.py* file we saved in the same folder, as we will use the functions in it later.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv")

Python returns the following error if we try this command

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf4 in position 1: invalid continuation byte

Pandas library is unable to decode the data using the default option. When we import the data, we must use the encoding option *ISO-8859-1* in order for Python to correctly interpret the data in the csv.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")

Now we import the csv as a data frame without error. There exist two columns that don’t actually have data and some rows with NA values. We will drop these.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")
6. **del** data["Unnamed: 34"]
7. **del** data["Unnamed: 35"]
9. # Drop countries with na values
10. data = data.dropna(thresh = 34)

Now that we have imported data, we need to clean the data. It is impossible to teach the basics of cleaning data in a short section. Your ability to clean data is dependent upon your ability to find solutions on the spot. This requires practice. Almost any data that you import will need to be cleaned. There will be missing values, string characters, etc… You will likely want to convert the data value to floats. Below, we remove symbols that cannot be read as values and convert these to floats.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")
6. **del** data["Unnamed: 34"]
7. **del** data["Unnamed: 35"]
9. # Drop countries with na values
10. data = data.dropna(thresh = 34)
12. **for** x **in** data:
13. **try**:
14. data[x] = data[x].str.replace(r'[$,]', '').astype('float')
16. **except**:
17. **print**(x, "not converted:", data[x].dtype)

Output:

Symbols not in CountryID float64

Symbols not in Country Name object

Symbols not in WEBNAME object

Symbols not in Region object

Symbols not in World Rank float64

Symbols not in Region Rank float64

Symbols not in 2017 Score float64

Symbols not in Property Rights float64

Symbols not in Judical Effectiveness float64

Symbols not in Government Integrity float64

Symbols not in Tax Burden float64

Symbols not in Gov't Spending float64

Symbols not in Fiscal Health float64

Symbols not in Business Freedom float64

Symbols not in Labor Freedom float64

Symbols not in Monetary Freedom float64

Symbols not in Trade Freedom float64

Symbols not in Investment Freedom float64

Symbols not in Financial Freedom float64

Symbols not in Tariff Rate (%) object

Symbols not in Income Tax Rate (%) float64

Symbols not in Corporate Tax Rate (%) float64

Symbols not in Tax Burden % of GDP float64

Symbols not in Country object

Symbols not in Unemployment (%) object

A number of objects have failed to convert because they lack the symbols that we deleted. Of these, only two have numerical values: Tax Burden % and Unemployment (%). We need to drop values that are marked “n/a”. We will convert them to NaN, then drop these cells using the command from line 10.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")
6. **del** data["Unnamed: 34"]
7. **del** data["Unnamed: 35"]
9. # Drop countries with na values
10. data = data.dropna(thresh = 34)
12. **for** x **in** data:
13. **try**:
14. data[x] = data[x].str.replace(r'[$,]', '').astype('float')
16. **except**:
17. **print**("Symbols not in", x, data[x].dtype)
19. # Unemployment (%) and Tariff Rate (%) did not convert successfully due to n/a values
20. data["Unemployment (%)"] = data["Unemployment (%)"].replace(['n/a'], 'NaN').astype('float')
21. data["Tariff Rate (%)"]= data["Tariff Rate (%)"].replace(['n/a'], 'NaN').astype('float')
22. data = data.dropna(thresh = 34)

Now the data is cleaned and ready to process. In the next chapter, we will compile regression statistics. For now, we will use the classes we created in the previous chapter to compile statistics. We will make three dictionaries. One dictionary will hold summary statistics that can be derived using only one variable. The other statistics that require two variables, correlation and covariance, will use separate dictionaries. In order to create these, we remove variables that are composed of string objects. We also remove “CountryID” since these values have no significance for the object of study. We will use an instance of *Stats()* to fill these dictionaries with summary statistics.

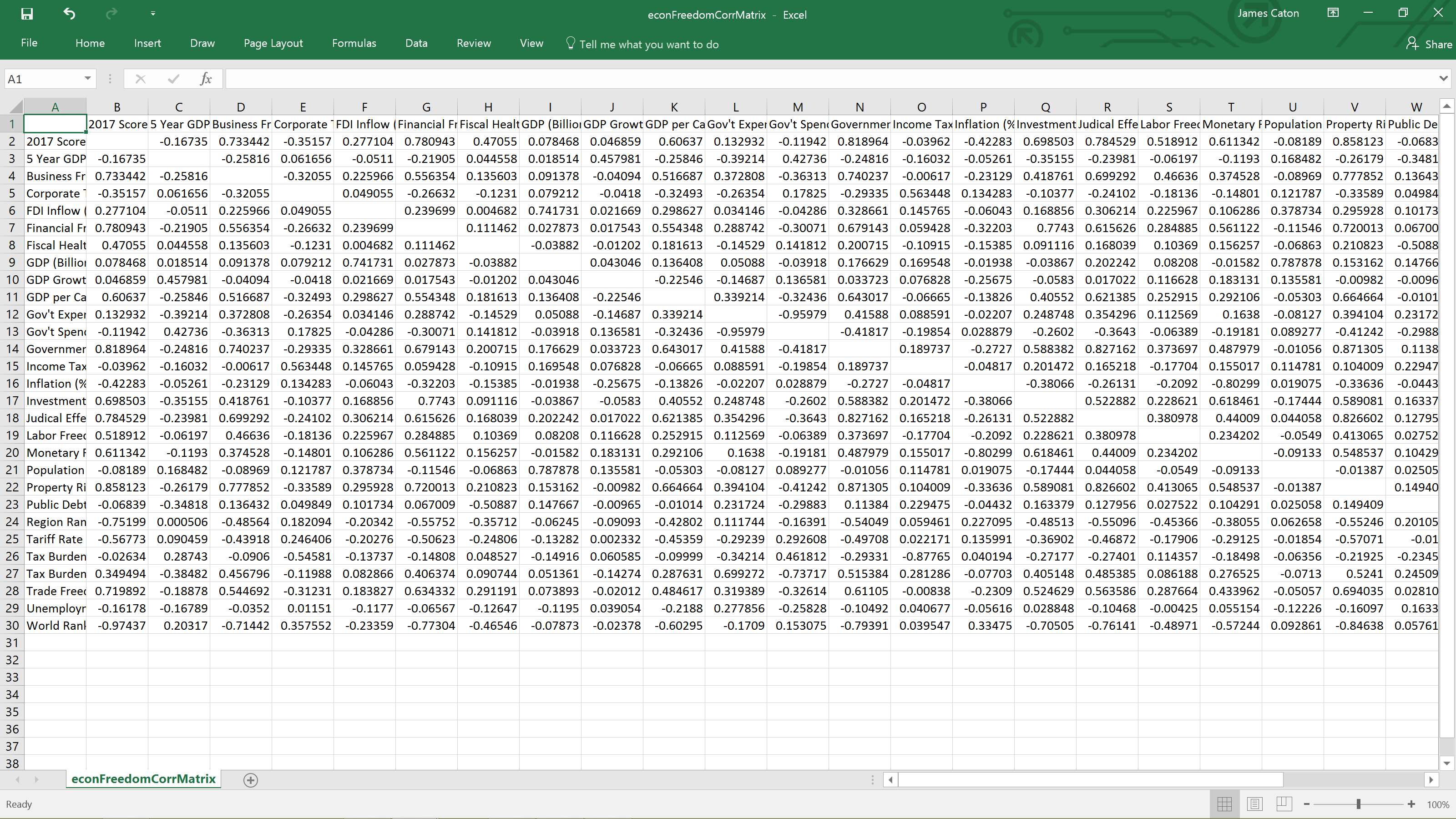
1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")
6. **del** data["Unnamed: 34"]
7. **del** data["Unnamed: 35"]
8. # Drop countries with na values
9. data = data.dropna(thresh = 34)
11. **for** x **in** data:
12. **try**:
13. data[x] = data[x].str.replace(r'[$,]', '').astype('float')
15. **except**:
16. **print**("Symbols not in", x, data[x].dtype)
18. # Unemployment (%) and Tariff Rate (%) did not convert successfully due to n/a values
19. data["Unemployment (%)"] = data["Unemployment (%)"].replace(['n/a'], 'NaN').astype('float')
20. data["Tariff Rate (%)"]= data["Tariff Rate (%)"].replace(['n/a'], 'NaN').astype('float')
21. data = data.dropna(thresh = 34)
23. stats = Stats()
24. statsDict = {}
25. corrDict = {}
26. covDict = {}
28. matData = data
29. **del** matData["CountryID"]
30. **del** matData["Country"]
31. **del** matData["Country Name"]
32. **del** matData["WEBNAME"]
33. **del** matData["Region"]

Now that these have been removed, we are ready to fill the dictionaries with summary statistics. We will use two for-loops. The first for-loop will be used to fill the *statsDict* with summary statistics requiring only one variable. The second for-loop, which is within the first for-loop, is used to create the *corrDict* and *covDict*. These latter two dictionaries are each a dictionary of dictionaries. These are called by entering the two variables whose correlation or covariance you would like to see. For example, if I enter *covDict[“2017 Score”][“Property Rights”]* after *covDict* has been constructed, I will call the covariance of these two variables. The two for-loops cycle through each variable twice to create a matrix of covariance and correlation values.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")
6. **del** data["Unnamed: 34"]
7. **del** data["Unnamed: 35"]
9. # Drop countries with na values
10. data = data.dropna(thresh = 34)
12. **for** x **in** data:
13. **try**:
14. data[x] = data[x].str.replace(r'[$,]', '').astype('float')
16. **except**:
17. **print**("Symbols not in", x, data[x].dtype)
19. # Unemployment (%) and Tariff Rate (%) did not convert successfully due to n/a values
20. data["Unemployment (%)"] = data["Unemployment (%)"].replace(['n/a'], 'NaN').astype('float')
21. data["Tariff Rate (%)"]= data["Tariff Rate (%)"].replace(['n/a'], 'NaN').astype('float')
22. data = data.dropna(thresh = 34)
24. stats = Stats()
25. statsDict = {}
26. corrDict = {}
27. covDict = {}
29. matData = data
30. **del** matData["CountryID"]
31. **del** matData["Country"]
32. **del** matData["Country Name"]
33. **del** matData["WEBNAME"]
34. **del** matData["Region"]
36. **for** x **in** matData:
37. vec = data[x].values.tolist()
38. statsDict[x] = {}
39. statsDict[x]["mean"] = stats.mean(vec)
40. statsDict[x]["median"] = stats.median(vec)
41. statsDict[x]["variance"] = stats.variance(vec, True)
42. statsDict[x]["standard deviation"] = stats.stdev(vec, True)
43. statsDict[x]["skewness"] = stats.skew(vec,  True)
44. statsDict[x]["kurtosis"] = stats.kurt(vec)
45. covDict[x] = {}
46. corrDict[x] = {}
47. **for** y **in** data:
48. vec2 = data[y].values.tolist()
49. **if** x != y:
50. covDict[x][y] = statsObj.covariance(vec, vec2, True)
51. corrDict[x][y] = statsObj.correlation(vec, vec2, True)

Finally, if we want to see these tables in a convenient format (i.e., Figure 2.2), we need to export the dictionaries to a csv. First, we convert the dictionaries into pandas data frames using *pd.DataFrame()*. Once the data frames are created, use the command *dataframe.to\_csv(path)* to create csv files in the desired location. In line 61, we also save a csv of the cleaned data for use in the next section.

1. # economicFreedomStats.py
2. **import** pandas as pd
3. **from** stats **import** \*
5. data = pd.read\_csv("index2017\_data.csv", encoding = "ISO-8859-1")
6. **del** data["Unnamed: 34"]
7. **del** data["Unnamed: 35"]
9. # Drop countries with na values
10. data = data.dropna(thresh = 34)
12. **for** x **in** data:
13. **try**:
14. data[x] = data[x].str.replace(r'[$,]', '').astype('float')
16. **except**:
17. **print**("Symbols not in", x, data[x].dtype)
19. # Unemployment (%) and Tariff Rate (%) did not convert successfully due to n/a values
20. data["Unemployment (%)"] = data["Unemployment (%)"].replace(['n/a'], 'NaN').astype('float')
21. data["Tariff Rate (%)"]= data["Tariff Rate (%)"].replace(['n/a'], 'NaN').astype('float')
22. data = data.dropna(thresh = 34)
24. stats = Stats()
25. statsDict = {}
26. corrDict = {}
27. covDict = {}
29. matData = data
30. **del** matData["CountryID"]
31. **del** matData["Country"]
32. **del** matData["Country Name"]
33. **del** matData["WEBNAME"]
34. **del** matData["Region"]
36. **for** x **in** matData:
37. vec = data[x].values.tolist()
38. statsDict[x] = {}
39. statsDict[x]["mean"] = stats.mean(vec)
40. statsDict[x]["median"] = stats.median(vec)
41. statsDict[x]["variance"] = stats.variance(vec, True)
42. statsDict[x]["standard deviation"] = stats.stdev(vec, True)
43. statsDict[x]["skewness"] = stats.skew(vec,  True)
44. statsDict[x]["kurtosis"] = stats.kurt(vec)
45. covDict[x] = {}
46. corrDict[x] = {}
47. **for** y **in** data:
48. vec2 = data[y].values.tolist()
49. **if** x != y:
50. covDict[x][y] = statsObj.covariance(vec, vec2, True)
51. corrDict[x][y] = statsObj.correlation(vec, vec2, True)
53. # create pandas DataFrames for viewing purposes
54. statsDictDF = pd.DataFrame(statsDict)
55. covDF = pd.DataFrame(covDict)
56. corrDF = pd.DataFrame(corrDict)
57. # output to CSV
58. statsDictDF.to\_csv("econFreedomStatsbyCategory.csv")
59. covDF.to\_csv("econFreedomCovMatrix.csv")
60. corrDF.to\_csv("econFreedomCorrMatrix.csv")
61. data.to\_csv("cleanedEconFreedomIndex.csv")



Correlation Matrix in CSV Format

**Figure 2.2**

* 1. **Using *pandas\_datareader***
  2. **Visualizing Data**

**[7. Building a Regression Analysis Function](#TableOfContents)**

Having built our statistics functions, we are now ready to build a function for regression analysis. We will start by building the most basic regression function that we can. We will use linear algebra to estimate parameters that minimize the sum of the squared errors. This is an ordinary least squares regression.

An OLS regression with one exogenous variable takes the form.

We merge the error term, which represents bias in the data, with alpha to yield the constant, . Each estimate of an individual point takes the form

Ideally, we want to form a prediction where, on average, the right-hand side of the equation yields the correct value on the left-hand side. When we perform an OLS regression, we form a predictor that minimizes the sum of the distance between each predicted value and the observed value drawn from the data. For example, if the prediction for a particular value of y is 8, and the actual value is 10, the error of the prediction is -2 and the squared error is 8

To find the function that minimizes the sum squared errors, we will use matrix algebra, also known as linear algebra. If you are unfamiliar with linear algebra, this is ok, we will use the numpy to perform matrix operations. For clarity, we will review the linear algebra functions that we will use with simple examples.

**Linear Algebra for OLS**

We solve the following function for an vector of beta values (, constants whose values represent estimates of the effect on the selected endogenously generate variable by the variables believed to cause this change. The matrix also includes a vector of 1s that are used to estimate the constant .

= (

= Observations for Endogenous Variable

= Observations for Exogenous Variables

= X-transpose

(

In reviewing the linear equation for estimating **,** we confront two unique operations worth understanding. Included in these are some key concepts in linear algebra, including the identity matrix and linear independence. The best way to understand these concepts is by working with some sample vectors. Consider the matrix consisting of vectors .We mustcheck that these vectors are linearly independent. We do this by joining with an identity matrix and thus create:

We can transform this such that:

Since

Let us solve for using the following vectors for X.

The identity matrix for a 3 x 3 matrix is:

We combine these

If we perform row operations on to transform in into,then wewill be transformed into

**r2 – 4r1**

**r3 – 6r1**

**r2 ⬄ r3**

**r2 / -4**

**r3 + 7r2**

**r1 – 2r2 – r3**

By transforming matrix into matrix we confirm that the vectors comprising are independent, meaning that one cannot be formed from the combination and or transformation of the others.A fundamental assumption of regression analysis is that data generated from factors believed to determine the y-values are independent of one another. Due to the nature of data generation and collection, data may seem to be independent as the matrix formed can be reduced to an identity matrix. There are instruments that we can use to tease out endogenous effects, some of which we will cover in this chapter.

**Linear Algebra in NumPy**

We can check this using linear algebra functions in numpy. We start by creating vertical numpy arrays that we will transform into vectors in the second step.

1. #invertMatrix.py
2. **import** numpy as np
3. #create vertical array
4. x1 = np.array([1,4,6])
5. x2 = np.array([2,1,8])
6. x3 = np.array([1,5,6])
8. **print**("Array 1\n", x1)
9. **print**("Array 2\n", x2)
10. **print**("Array 3\n", x3)

Output:

Array 1

[1 4 6]

Array 2

[2 1 8]

Array 3

[1 5 6]

We want to create vertical rather than horizontal arrays. Following the *array()* command, tell python to create a vertical, rather than a horizontal array using the command *[:,None]*

1. #invertMatrix.py
2. **import** numpy as np
3. #create vertical array
4. x1 = np.array([1,4,6])[:, None]
5. x2 = np.array([2,1,8])[:, None]
6. x3 = np.array([1,5,6])[:, None]
8. **print**("Array 1\n", x1)
9. **print**("Array 2\n", x2)
10. **print**("Array 3\n", x3)

Output:

Array 1

[[1]

[4]

[6]]

Array 2

[[2]

[1]

[8]]

Array 3

[[1]

[5]

[6]]

Join them using the *concatenate()* function. We define *axis=1* to stack the vertical vectors from left to right.

1. #invertMatrix.py
2. **import** numpy as np
3. #create vertical array
4. x1 = np.array([1,4,6])[:, None]
5. x2 = np.array([2,1,8])[:, None]
6. x3 = np.array([1,5,6])[:, None]
7. #transforms arrays into matrix arrays
8. x1 = np.matrix(x1)
9. x2 = np.matrix(x2)
10. x3 = np.matrix(x3)
12. **print**("Array 1\n", x1)
13. **print**("Array 2\n", x2)
14. **print**("Array 3\n", x3)

Output:

Array 1

[[1]

[4]

[6]]

Array 2

[[2]

[1]

[8]]

Array 3

[[1]

[5]

[6]]

Next, transform these arrays into matrix vectors using *matrix()*.

1. #invertMatrix.py
2. **import** numpy as np
3. #create vertical array
4. x1 = np.array([1,4,6])[:, None]
5. x2 = np.array([2,1,8])[:, None]
6. x3 = np.array([1,5,6])[:, None]
7. #transforms arrays into matrix arrays
8. x1 = np.matrix(x1)
9. x2 = np.matrix(x2)
10. x3 = np.matrix(x3)
11. #join vectors into a 3x3 matrix
12. X = np.concatenate((x1, x2, x3), axis=1)
14. **print**("Array 1\n", x1)
15. **print**("Array 2\n", x2)
16. **print**("Array 3\n", x3)
17. **print**("Matrix of x1, x2, x3\n", X)

Finally, we can invert the matrix that we have made using *getI()*.

1. #invertMatrix.py
2. **import** numpy as np
3. #create vertical array
4. x1 = np.array([1,4,6])[:, None]
5. x2 = np.array([2,1,8])[:, None]
6. x3 = np.array([1,5,6])[:, None]
7. #transforms arrays into matrix arrays
8. x1 = np.matrix(x1)
9. x2 = np.matrix(x2)
10. x3 = np.matrix(x3)
11. #join vectors into a 3x3 matrix
12. X = np.concatenate((x1, x2, x3), axis=1)
13. invertX = X.getI()
15. **print**("Array 1\n", x1)
16. **print**("Array 2\n", x2)
17. **print**("Array 3\n", x3)
18. **print**("Matrix of x1, x2, x3\n", X)
19. **print**("Inverted Matrix of x1, x2, x3\n", invertX)

Output:

[[1]

[4]

[6]]

[[2]

[1]

[8]]

[[1]

[5]

[6]]

[[1 2 1]

[4 1 5]

[6 8 6]]

[[ -8.50000000e+00 -1.00000000e+00 2.25000000e+00]

[ 1.50000000e+00 -6.08487619e-17 -2.50000000e-01]

[ 6.50000000e+00 1.00000000e+00 -1.75000000e+00]]

These values are not rounded, so interpretation of the inverted matrix is not as clear as could be. We use the *round()* to round values to two places.

1. #invertMatrix.py
2. **import** numpy as np
3. #create vertical array
4. x1 = np.array([1,4,6])[:, None]
5. x2 = np.array([2,1,8])[:, None]
6. x3 = np.array([1,5,6])[:, None]
7. #transforms arrays into matrix arrays
8. x1 = np.matrix(x1)
9. x2 = np.matrix(x2)
10. x3 = np.matrix(x3)
12. #join vectors into a 3x3 matrix
13. X = np.concatenate((x1, x2, x3), axis=1)
14. invertX = X.getI()
15. invertX = np.round(invertX, 2)
17. **print**("Array 1\n", x1)
18. **print**("Array 2\n", x2)
19. **print**("Array 3\n", x3)
20. **print**("Matrix of x1, x2, x3\n", X)
21. **print**("Inverted Matrix of x1, x2, x3\n", invertX)

Output:

Array 1

[[1]

[4]

[6]]

Array 2

[[2]

[1]

[8]]

Array 3

[[1]

[5]

[6]]

Matrix of x1, x2, x3

[[1 2 1]

[4 1 5]

[6 8 6]]

Inverted Matrix of x1, x2, x3

[[-8.5 -1. 2.25]

[ 1.5 -0. -0.25]

[ 6.5 1. -1.75]]

**Building a Regression Function**

Now that we have learned the necessary operations, we can create a regression function. Recall that we estimate the beta parameters for each variable with the equation

= (

In order to estimate the parameters, we will need to import data, define the dependent variable and independent variables, and transform these into matrix objects. We will use one py file to write a regression function and another to write the script that calls the regression function. Let’s start by importing the data that we created already.

1. #econFreedomRegression.py
2. **import** pandas as pd
4. data = pd.DataFrame.from\_csv("cleanedEconFreedomIndex.csv")
5. **print**(data)

After importing the data, we print it to be sure that we have imported correctly. The first part of the results should match the output below.

Output:

World Rank Region Rank 2017 Score Property Rights \ …

1 65.0 30.0 64.4 54.0

2 172.0 14.0 46.5 38.2

3 165.0 41.0 48.5 36.4

4 156.0 26.0 50.4 32.4

5 33.0 19.0 70.3 55.5

6 5.0 4.0 81.0 81.7

7 30.0 17.0 72.3 86.0

8 68.0 15.0 63.6 50.5

…

Now that we have confirmed we have the correct dataset, you may delete the print command from line 5.

Next we will create the regression.py file. This will contain the regression program that we will call from econFreedomRegression.py. For now, import pandas and build the class as demonstrated below.

1. #regression.py
2. **from** stats **import** \*
4. **class** Regression():
5. **def** \_\_init\_\_(self):
6. self.x = 0
7. self.stats = Stats()
8. self.regDict = {}
10. **def** regress(self, regName, data, yName, betaNames, minVal=0, maxVal = None,
11. logVars = None, firstDiff = False, numLags = 0):
12. self.regName = regName
13. self.regDict[self.regName] = {}
14. self.minVal = minVal
15. **if** maxVal == None:
16. self.maxVal = len(data[yName])
17. **else**:
18. self.maxVal = maxVal
19. self.data = data
20. self.yName = yName
21. self.betaNames = betaNames

We start by importing pandas and the stats py file that we have already saved in the same folder. We create two functions inside of the *Regression* class. First is the *\_\_init\_\_* function. This will create an instance of *Stats* that will be called later. Second is the *regress* function. This is our main function, from which all the necessary steps for preparing data and running a regression will be called. The *regress* function passes several objects. First is regName, which will is a string that be used later to call the results of the regression. Data is the pandas data frame used for the regression. Next are the names of the variables we wish to regress: *yName* is the name of the dependent variable and *xNames* is an array that includes the names of variables that we wish to regress. *minVal* and *maxVal* are the starting and ending index values for the regression. We include also *logVars* and *firstDiff* which will be used later to change the functional forms of the regression. Finally, we include numLags which can be used to tease out lagged effects in time-series regressions. We will not add these extensions until we build the basic regression function.

When we first call the regress function, we call the objects that we have passed and make sure that

1. #econFreedomRegression.py
2. **import** pandas as pd
3. **import** regression as reg
5. data = pd.DataFrame.from\_csv("cleanedEconFreedomIndex.csv")
6. regression = reg.Regression()
7. **print**(regression)

Output:

<regression.Regression object at 0x00000053EDD9EF98>

A standard OLS regression assumes that the equation it is estimating will include a constant. We must therefore include a space for a constant in our data that we will use to estimate parameter values. To do this, we add a column of ones that will be used to estimate a constant value for our equation. This column of ones is identified by the column name, “Constant”.

1. #regression.py
2. **from** stats **import** \*
3. **import** pandas as pd
5. **class** Regression():
6. **def** \_\_init\_\_(self):
7. self.stats = Stats()
8. self.regDict = {}
10. **def** regress(self, regName, data, yName, betaNames, includeConstant = True,
11. minVal = 0, maxVal=None, logVars = None, firstDiff = False,
12. numLags = 0):
13. self.regName = regName
14. self.regDict[self.regName] = {}
15. self.minVal = minVal
16. **if** maxVal == None:
17. self.maxVal = len(data[yName])
18. **else**:
19. self.maxVal = maxVal
20. self.data = data
21. self.yName = yName
22. self.betaNames = betaNames
23. **if** includeConstant:
24. self.addConstant()
25. self.betaNames.append("Constant")
27. **def** addConstant(self):
28. self.data["Constant"] = pd.Series([1] \* self.maxVal,
29. index = self.data.index)

To see the effect of this addition, we can print the data after we have call the regression function from our object that is an instance of the *Regression* class. We will choose to print the “Constant” column.

1. #econFreedomRegression.py
2. **import** pandas as pd
3. **import** regression as reg
5. data = pd.DataFrame.from\_csv("cleanedEconFreedomIndex.csv")
6. yVar = "5 Year GDP Growth Rate (%)"
7. xVars = ["Inflation (%)", "Gov't Expenditure % of GDP ", "2017 Score",
8. "Population (Millions)"]
9. regression = reg.Regression()
10. regression.regress("Test", data, yVar, xVars)
11. **print**(regression.data["Constant"])

Output:

1 1

2 1

3 1

4 1

5 1

6 1

7 1

8 1

9 1

10 1

…

170 1

171 1

172 1

173 1

174 1

175 1

176 1

177 1

178 1

179 1

181 1

182 1

185 1

Name: Constant, dtype: int64

Now we will build the core of the regression. Using the numpy command *matrix*, we will create matrices and perform operations on them using operations that we have already used. We will first create the y and X matrices. The y matrix will hold our dependent variable. The X matrix will hold the independent variables and the “Constant” array that we created. We must also create a transpose of the X matrix. With these, we will be able to implement an OLS regression.

1. #regression.py
2. **from** stats **import** \*
3. **import** pandas as pd
4. **import** numpy as np
6. **class** Regression():
7. **def** \_\_init\_\_(self):
8. self.stats = Stats()
9. self.regDict = {}
11. **def** regress(self, regName, data, yName, betaNames, includeConstant = True,
12. minVal = 0, maxVal=None, logVars = None, firstDiff = False,
13. numLags = 0):
14. self.regName = regName
15. self.regDict[self.regName] = {}
16. self.minVal = minVal
17. **if** maxVal == None:
18. self.maxVal = len(data[yName])
19. **else**:
20. self.maxVal = maxVal
21. self.data = data
22. self.yName = yName
23. self.betaNames = betaNames
24. **if** includeConstant:
25. self.addConstant()
26. self.buildMatrices()
28. **def** addConstant(self):
29. self.data["Constant"] = pd.Series([1] \* self.maxVal,
30. index = self.data.index)
31. self.betaNames.append("Constant")
33. **def** buildMatrices(self):
34. **print**(""" Beta = (X'X)^(-1)X'Y """)
35. # Convert dataframes to matrices
36. self.matrixArray = []
37. self.y = np.matrix(pd.DataFrame(self.data[self.yName]\
38. [self.minVal:self.maxVal]))
39. # create standard array of X values
40. **for** name **in** self.betaNames:
41. self.matrixArray.append(self.data[name][self.minVal:self.maxVal])
42. # Transform into matrix, must be Transposed to create correct matrix
43. self.X = np.matrix(self.matrixArray).getT()
44. # Transpose again to get X^T
45. self.Xtranspose = self.X.getT()
47. #(X'X)^-1
48. self.Betas = np.matmul(self.Xtranspose, self.X)
49. self.Betas = self.Betas.getI()
50. # X'Y
51. self.Xty = np.matmul(self.Xtranspose, self.y)
52. self.Betas = np.matmul(self.Betas, self.Xty)
54. # y-hat = X\*Betas
55. self.yhat = np.matmul(self.X, self.Betas)
56. self.betasValues = pd.DataFrame(self.Betas, index = self.betaNames,
57. columns = ["Beta Values"])

From the *econFreedomRegression.py* file, lets execute the *regress* function that we have extended. Executing it will generate the data frame of beta values. We will print these in line 11 to make sure that the calcuations have been correctly recorded. Thereafter, we will delete this print command in line 11.

1. #econFreedomRegression.py
2. **import** pandas as pd
3. **import** regression as reg
5. data = pd.DataFrame.from\_csv("cleanedEconFreedomIndex.csv")
6. yVar = "5 Year GDP Growth Rate (%)"
7. xVars = ["Inflation (%)", "Gov't Expenditure % of GDP ", "2017 Score",
8. "Population (Millions)"]
9. regression = reg.Regression()
10. regression.regress("Test", data, yVar, xVars)
11. **print**(regression.betasValues)

Output:

Beta = (X'X)^(-1)X'Y

Beta Values

Inflation (%) 0.022379

Gov't Expenditure % of GDP -0.051886

2017 Score 0.077588

Population (Millions) 0.003269

Constant 8.836251

We have calculated beta values for each variable, meaning that we estimated the average effect of a change in each independent variable upon the dependent variable. While this is useful, we have not yet estimated the significance of these estimations; neither have we determined the explanatory power of our particular regression.

Our regression has estimated predicted values for our dependent variable given the values of the independent variables for each observation. Together, these estimations for an array of predicted values that we will refer to as . We will refer to individual predicted values as . We will also refer to the mean value of observations of our dependent variable as and individual observed values of our dependent variable as . These values will be use to estimate the sum of squares due to regression (SSR), sum of squared errors (SSE), and the total sum of squares (SST). By comparing the estimated , the observed , and the mean , we will be able to estimate the accuracy of each estimated value and the regression as a whole.

We define these values as follows:

It happens that the sum of the squared distances between the estimated values and mean of observed values and the squared distances between the observed and estimated values add up to the sum of the squared distances between the observed values and the mean of observed values. We indicate this as:

We estimate these below in the *errSums* function, which is passed in the *calcRegStats* function.

1. #regressionTest.py

...

1. **def** regress(self, regName, data, yName, betaNames, includeConstant = True,
2. minVal = 0, maxVal=None, logVars = None, firstDiff = False,
3. numLags = 0):
4. self.regName = regName
5. self.regDict[self.regName] = {}
6. self.minVal = minVal
7. **if** maxVal == None:
8. self.maxVal = len(data[yName])
9. **else**:
10. self.maxVal = maxVal
11. self.data = data
12. self.yName = yName
13. self.betaNames = betaNames
14. **if** includeConstant:
15. self.addConstant()
16. self.buildMatrices()
17. **print**("Calculating Regression Stats")
18. self.calcRegStats()
20. **def** addConstant(self):
21. self.data["Constant"] = pd.Series([1] \* self.maxVal,
22. index = self.data.index)
23. self.betaNames.append("Constant")
25. **def** buildMatrices(self):
26. **print**(""" Beta = (X'X)^(-1)X'Y """)
27. # Convert dataframes to matrices
28. self.matrixArray = []
29. self.y = np.matrix(pd.DataFrame(self.data[self.yName]\
30. [self.minVal:self.maxVal]))
31. # create standard array of X values
32. **for** name **in** self.betaNames:
33. self.matrixArray.append(self.data[name][self.minVal:self.maxVal])
34. # Transform into matrix, must be Transposed to create correct matrix
35. self.X = np.matrix(self.matrixArray).getT()
36. # Transpose again to get X^T
37. self.Xtranspose = self.X.getT()
39. #(X'X)^-1
40. self.Betas = np.matmul(self.Xtranspose, self.X)
41. self.Betas = self.Betas.getI()
42. # X'Y
43. self.Xty = np.matmul(self.Xtranspose, self.y)
44. self.Betas = np.matmul(self.Betas, self.Xty)
46. # y-hat = X\*Betas
47. self.yhat = np.matmul(self.X, self.Betas)
48. self.betasValues = pd.DataFrame(self.Betas, index = self.betaNames,
49. columns = ["Beta Values"])
51. **def** calcRegStats(self):
52. # calculate sse, sst, ssr
53. self.errSums()
55. **def** errSums(self):
56. self.ssrArray = []
57. self.sseArray = []
58. self.sstArray = []
59. meany = self.stats.mean(self.y)
60. **for** i **in** range(len(self.y)):
61. # ssr is sum of squared distances between the estimated y values,
62. # y-hat, and the average y-value
63. self.ssrArray.append((self.yhat[i] - meany) \*\* 2)
64. # sse is the sum of squared distances between observed y values
65. # and estimated y-values
66. self.sseArray.append((self.y[i] - self.yhat[i]) \*\* 2)
67. # sst is the sum of squared distances between observed y values
68. # and estimated y-values
69. self.sstArray.append((self.y[i] - meany) \*\* 2)
70. # call item 0 to avoid error that results from using numpy matrix
71. self.ssr = self.stats.total(self.ssrArray)
72. self.sst = self.stats.total(self.sstArray)
73. self.sse = self.stats.total(self.sseArray)
74. **print**(self.ssr, self.sst, self.sse)

If we execute the *econFreedomRegression.py* script, we will see that *self.ssr, self.sst, and self.sse* return values that are still in matrix form.

Output:

Beta = (X'X)^(-1)X'Y

Calculating Regression Stats

[[ 214.55196458]] [[ 1081.51976608]] [[ 866.9678015]]

The matrices actually return tuples, so we must choose the element in each that hold the results we seek to use. We correct this problem by attaching *.item(0)* to the total function that sums each array.

1. self.ssr = self.stats.total(self.ssrArray).item(0)
2. self.sst = self.stats.total(self.sstArray).item(0)
3. self.sse = self.stats.total(self.sseArray).item(0)
4. **print**(self.ssr, self.sst, self.sse)

Output:

Beta = (X'X)^(-1)X'Y

Calculating Regression Stats

214.5519645781984 866.9678015036704 1081.5197660818715

Now that these are correctly formatted, we can delete the *print()* command and continue building the regression function.

Next we will use the statistics that we have calculated to build the mean squared error (MSE), the square root of the mean squared error, R2, and F-stat. We will also generate a covariance matrix from which we will draw the variance of each beta value.

We can think of the estimate of the MSE as the variance of the estimator generated by the We estimate the variance of the error term of the estimator for the dependent variable.

The variance term will be used to help us calculate other values. First we estimate the square root of the mean squared error. Since the mean squared error is the variance of the estimator, this means we simply take the square root the variance term.

The square-root of the MSE provides a more readily interpretable estimate of the estimator variance, showing the average distance of predicted values from actual values, corrected for the number of independent variables.

We also estimate the R2 value. This value indicates the explanator power of the regression

This compares the average squared distance between the predicted values and the average value against the average squared distance between observed values and average values. Ordinary least squares regression minimizes the squared distance between the predicted value and the average value. If values are perfectly predicted, then the SSR would equal the SST. Usually, the SSR is less than the SST. It will never be greater than the SST.

Finally we calculate the F-statistic, commonly referred to as the F-stat. The F-statistic tests the likelihood of whether or not the values of our estimated parameters are all zero. We check the difference between the SST and SSE divided by the number of independent variables used in the regression less one. We divide this value by the mean squared error.

1. self.fstat = ((self.sst - self.sse) / (lostDegreesOfFreedom - 1)) /\
2. (self.varEstimator)

…

1. #regressionTest.py

...

2. **def** calcRegStats(self):
3. # calculate sse, sst, ssr
4. self.errSums()
5. self.calcStatsFromSummedSquares()
6. self.genRegCovMatrix()
8. **def** errSums(self):
9. self.ssrArray = []
10. self.sseArray = []
11. self.sstArray = []
12. meany = self.stats.mean(self.y)
13. **for** i **in** range(len(self.y)):
14. # ssr is sum of squared distances between the estimated y values,
15. # y-hat, and the average y-value
16. self.ssrArray.append((self.yhat[i] - meany) \*\* 2)
17. # sse is the sum of squared distances between observed y values
18. # and estimated y-values
19. self.sseArray.append((self.y[i] - self.yhat[i]) \*\* 2)
20. # sst is the sum of squared distances between observed y values
21. # and estimated y-values
22. self.sstArray.append((self.y[i] - meany) \*\* 2)
23. # call item 0 to avoid error that results from using numpy matrix
24. self.ssr = self.stats.total(self.ssrArray).item(0)
25. self.sst = self.stats.total(self.sstArray).item(0)
26. self.sse = self.stats.total(self.sseArray).item(0)
28. **def** calcStatsFromSummedSquares(self):
29. lostDegreesOfFreedom = len(self.betasValues)
30. degreesOfFreedom = (self.maxVal – self.minVal) - len(self.betasValues)
31. self.varEstimator = self.sse / degreesOfFreedom
32. self.mse = self.varEstimator \*\* (1/2)
33. self.rsq = self.ssr / self.sst
34. self.fstat = ((self.sst - self.sse) / (lostDegreesOfFreedom - 1)) /\
35. (self.varEstimator)
37. **def** genRegCovMatrix(self):
38. self.covMatrix = np.matmul(self.X.getT(), self.X).getI()
39. **if** self.varEstimator != None:
40. covMatrix = float(self.varEstimator) \* self.covMatrix
41. self.covMatrix = pd.DataFrame(self.covMatrix, columns = self.betaNames,
42. index = self.betaNames)

regression. a beta parameter by generating a covariance matrix whose diagonal traveling down and to the right consists of variance values for each estimated parameter.

1. #econFreedomRegression.py
2. **import** pandas as pd
3. **import** regression as reg
5. data = pd.DataFrame.from\_csv("cleanedEconFreedomIndex.csv")
6. yVar = "5 Year GDP Growth Rate (%)"
7. xVars = ["Inflation (%)", "Gov't Expenditure % of GDP ", "2017 Score",
8. "Population (Millions)"]
9. regression = reg.Regression()
10. regression.regress("Test", data, yVar, xVars)
11. **print**(regression.betasValues)
12. **print**("variance estimator:", regression.varEstimator)
13. **print**("mse:", regression.mse)
14. **print**("r-squared:", regression.rsq)
15. **print**("f-stat:",regression.fstat)

Output:

Beta = (X'X)^(-1)X'Y

Calculating Regression Stats

Beta Values

Inflation (%) -0.031514

Gov't Expenditure % of GDP -0.084181

2017 Score -0.041576

Population (Millions) 0.002187

Constant 8.836251

variance estimator: 5.222697599419702

mse: 2.2853222091030623

r-squared: 0.1983800678516279

f-stat: 10.270169797024055

1. #regression.py
2. **from** stats **import** \*
3. **import** pandas as pd
4. **import** numpy as np
5. **from** decimal **import** Decimal
6. **from** scipy.stats **import** t

9. **class** Regression():
10. **def** \_\_init\_\_(self):
11. self.stats = Stats()
12. self.regDict = {}
14. **def** regress(self, regName, data, yName, betaNames, includeConstant = True,
15. minVal = 0, maxVal=None, logVars = None, firstDiff = False,
16. numLags = 0):
17. self.regName = regName
18. self.regDict[self.regName] = {}
19. self.minVal = minVal
20. **if** maxVal == None:
21. self.maxVal = len(data[yName])
22. **else**:
23. self.maxVal = maxVal
24. self.data = data
25. self.yName = yName
26. self.betaNames = betaNames
27. **if** includeConstant:
28. self.addConstant()
29. self.buildMatrices()
30. **print**("Calculating Regression Stats")
31. self.calcRegStats()
33. **def** addConstant(self):
34. self.data["Constant"] = pd.Series([1] \* self.maxVal,
35. index = self.data.index)
36. self.betaNames.append("Constant")
38. **def** buildMatrices(self):
39. **print**(""" Beta = (X'X)^(-1)X'Y """)
40. # Convert dataframes to matrices
41. self.matrixArray = []
42. self.y = np.matrix(pd.DataFrame(self.data[self.yName]\
43. [self.minVal:self.maxVal]))
44. # create standard array of X values
45. **for** name **in** self.betaNames:
46. self.matrixArray.append(self.data[name][self.minVal:self.maxVal])
47. # Transform into matrix, must be Transposed to create correct matrix
48. self.X = np.matrix(self.matrixArray).getT()
49. # Transpose again to get X^T
50. self.Xtranspose = self.X.getT()
52. #(X'X)^-1
53. self.Betas = np.matmul(self.Xtranspose, self.X)
54. self.Betas = self.Betas.getI()
55. # X'Y
56. self.Xty = np.matmul(self.Xtranspose, self.y)
57. self.Betas = np.matmul(self.Betas, self.Xty)
59. # y-hat = X\*Betas
60. self.yhat = np.matmul(self.X, self.Betas)
61. self.betasValues = pd.DataFrame(self.Betas, index = self.betaNames,
62. columns = ["Beta Values"])
64. **def** calcRegStats(self):
65. # calculate sse, sst, ssr
66. self.errSums()
67. self.calcStatsFromSummedSquares()
68. self.genRegCovMatrix()
70. self.standardErrors = {}
71. self.tStats = {}
72. self.pValues = {}
73. **print**("betaValues:\n", self.betasValues)
74. **for** name **in** self.betaNames:
75. self.standardErrors[name] =self.covMatrix[name][name] \*\* (1/2)
76. ## two tail t-stats
77. self.tStats[name] = self.betasValues.get\_value(name,"Beta Values")\
78. / self.standardErrors[name]
79. self.pValues[name] = (round(Decimal(t.sf(np.abs(self.tStats[name]),
80. self.maxVal - len(self.betasValues) + 1 ) \* 2  ), 4))
81. # Create DataFrames with Results
82. self.table = self.betasValues
83. self.table.index.name = "Endog Var: "+ self.regName
84. self.table['Std Errors'] = pd.Series(self.standardErrors)
85. self.table['t-stats'] = pd.Series(self.tStats)
86. self.table['p-values'] = pd.Series(self.pValues)
88. fitNames = ["Estimator Variance", "MSE", "F-stat", "R-squared"]
89. t2Height = len(fitNames)
90. self.table2= pd.DataFrame(pd.Series(["   "] \* t2Height),columns = ['fitNames'])
91. self.table2['fitStats'] = pd.Series(["   "] \* t2Height)
92. fitStats = [self.varEstimator,self.mse, self.fstat, self.rsq]
94. **for** i **in** range(t2Height):
95. self.table2['fitNames'][i] = fitNames[i]
96. self.table2['fitStats'][i] = fitStats[i]

99. **def** errSums(self):
100. self.ssrArray = []
101. self.sseArray = []
102. self.sstArray = []
103. meany = self.stats.mean(self.y)
104. **for** i **in** range(len(self.y)):
105. # ssr is sum of squared distances between the estimated y values,
106. # y-hat, and the average y-value
107. self.ssrArray.append((self.yhat[i] - meany) \*\* 2)
108. # sse is the sum of squared distances between observed y values
109. # and estimated y-values
110. self.sseArray.append((self.y[i] - self.yhat[i]) \*\* 2)
111. # sst is the sum of squared distances between observed y values
112. # and estimated y-values
113. self.sstArray.append((self.y[i] - meany) \*\* 2)
114. # call item 0 to avoid error that results from using numpy matrix
115. self.ssr = self.stats.total(self.ssrArray).item(0)
116. self.sst = self.stats.total(self.sstArray).item(0)
117. self.sse = self.stats.total(self.sseArray).item(0)
119. **def** calcStatsFromSummedSquares(self):
120. lostDegreesOfFreedom = len(self.betasValues)
121. degreesOfFreedom = (self.maxVal – self.minVal) - len(self.betasValues)
122. self.varEstimator = self.sse / degreesOfFreedom
123. self.mse = self.varEstimator \*\* (1/2)
124. self.rsq = self.ssr / self.sst
125. self.fstat = ((self.sst - self.sse) / (lostDegreesOfFreedom - 1)) /\
126. (self.varEstimator)
128. **def** genRegCovMatrix(self):
129. self.covMatrix = np.matmul(self.X.getT(), self.X).getI()
130. **if** self.varEstimator != None:
131. covMatrix = float(self.varEstimator) \* self.covMatrix
132. self.covMatrix = pd.DataFrame(self.covMatrix, columns = self.betaNames,
133. index = self.betaNames)
135. #regression.py
136. **from** stats **import** \*
137. **import** pandas as pd
138. **import** numpy as np
139. **from** decimal **import** Decimal
140. **from** scipy.stats **import** t

143. **class** Regression():
144. **def** \_\_init\_\_(self):
145. self.stats = Stats()
146. self.regDict = {}
148. **def** regress(self, regName, data, yName, betaNames, includeConstant = True,
149. minVal = 0, maxVal=None, logVars = None, firstDiff = False,
150. numLags = 0):
151. self.regName = regName
152. self.regDict[self.regName] = {}
153. self.minVal = minVal
154. **if** maxVal == None:
155. self.maxVal = len(data[yName])
156. **else**:
157. self.maxVal = maxVal
158. self.data = data
159. self.yName = yName
160. self.betaNames = betaNames
161. **if** includeConstant:
162. self.addConstant()
163. self.buildMatrices()
164. **print**("Calculating Regression Stats")
165. self.calcRegStats()
166. self.fillRegDict(self.regDict)

...

1. **def** exportResults(self, name):
2. self.table.to\_csv(name + "Results.csv")
3. self.table2.to\_csv(name + "Stats.csv")


7. **def** fillRegDict(self, dictionary):
8. dictionary["data"] = self.data
9. dictionary["yhat"] = self.yhat
10. dictionary["ssr"] = self.ssr
11. dictionary["sst"] = self.sst
12. dictionary["sse"] = self.sse
13. dictionary["fstat"]  = self.fstat
14. dictionary["pValues"] = pd.Series(self.pValues)
15. dictionary["tStats"] = pd.Series(self.tStats)
16. dictionary["betas"] = pd.DataFrame(self.betasValues)
17. dictionary["ehat"] = pd.Series(self.varEstimator)
18. dictionary["mse"] = self.mse
19. dictionary["rsquared"] = self.rsq
20. dictionary["covmatrix"] = self.covMatrix
21. dictionary["tables"] = {}
22. dictionary["tables"][1] = self.table
23. dictionary["tables"][2] = self.table2
24. #econFreedomRegression.py
25. **import** pandas as pd
26. **import** regressionTest as reg
28. data = pd.DataFrame.from\_csv("cleanedEconFreedomIndex.csv")
29. yVar = "5 Year GDP Growth Rate (%)"
30. xVars = ["Inflation (%)", "Gov't Expenditure % of GDP ", "2017 Score",
31. "Population (Millions)"]
32. regression = reg.Regression()
33. regression.regress("Test", data, yVar, xVars)
34. **print**("variance estimator:", regression.varEstimator)
35. **print**("mse:", regression.mse)
36. **print**("r-squared:", regression.rsq)
37. **print**("f-stat:",regression.fstat)
38. **print**(regression.regDict["tables"])

**[Chapter 8: Agent-based Models](#TableOfContents)**

You probably remember the first time that you played a video game. The television or computer monitor had a character or other object that responded to the press of buttons on a controller or keyboard. And don’t forget about the sound effects: BAM!-WOOM!-VOOM!-TICK!-BLIP!

You may not have realized it at the time, but you were interacting with an agent-based model. Characters in a game interact with an environment. This includes any other objects in the environment. This is accomplished by creating objects that are instances of different classes. In [chapter 4](#ClassesAndMethods) we created a class to serve as a statistical library. While this is useful, it does not exemplify broader applications that object-oriented programing (OOP) enables.

Classes can be used to create simulations where agents interact with one another. When you visit an arcade, it is not uncommon to see on some of the machines characters fighting with one another without any human players present! The fighting agents depending on aritificial intelligence (AI) to make decisions.

In this chapter we will create two agent-based models. The first is a role-playing game (RPG) where agents battle using a turn-based system. This will help you become familiar with most of the technical concepts required to understand an agent-based model. The second is a simulation of exchange that will be a useful prototype if you plan to create an agent-based model of economic activity. In the second model, we will organize data for visualization.

**a. Create Your Own Role-playing Game!**

You are about to create your own RPG. This game will exemplify fundamental concepts used to create many different games. You will create all of the functions for this game. When you are finished, you may decided that you want to develop the game further. If you do, we recommend that you familiarize yourself with the [pygame](https://www.pygame.org/wiki/GettingStarted) library.

Creating a game or a model using class requires planning of structure. This RPG will include agents that fight one another. These agents may be controlled by a human or by AI, which we refer to as *computer* in the script. The game will include two classes: Agent and RPG. The RPG class will be used to initiate and structure both individual melees and a tournament. The agent class will be used to structure agent decisions and define agent attributes.

We will begin by defining the RPG class. We will call an instance of this class to create and play the game. For now, we are interested only in how many humans are playing the game. We will create a dictionary that holds agents from the game and also create a list of ids that identify which players are human.

1. #RPG.py
2. **class** RPG():
4. **def** \_\_init\_\_(self, num\_humans):
5. self.num\_humans= num\_humans
6. self.agent\_dict = {}
7. self.human\_ids = []

We will return to the RPG class, but first we must create the game’s agents. We must identify the minimum requirement for the agent class. To interact with the game, agents must be able to access the instance of the *rpg* class that coordinates the game. Agents will require attributes that define their fitness. These include *strength*, *defense*, and *magic*. Agent health is defined as *hp* and their ability to cast a spell is requires the use of 1 *mp*. We will also include maximum values for *hp* and *mp*. We also need to need an identifier for each agent which will be defined as id\_*num*. We will also identify whether or not the agent is *alive* as well as the *record* of wins and losses.

1. #agent.py
3. **class** Agent():
5. **def** \_\_init\_\_(self, rpg, hp, mp, strength, defense, magic, id\_num):
6. self.rpg = rpg
7. self.id\_num = id\_num
8. self.alive = True
9. self.record = {"Wins": 0, "Losses": 0}
10. self.strength = strength
11. self.hp = hp
12. self.max\_hp = hp
13. self.mp = mp
14. self.max\_mp = mp
15. self.defense = defense
16. self.magic = magic

When an instance of the agent class is created, the *\_\_init\_\_* method is automatically executed. We can now enable the RPG class to create agents using the agent class. Below we import the agent.py with the command *import agent*. The file should be saved in the same directory as *rpg.py*.

After importing *agent*, create a method called *create\_players()*. This will be called unde the \_\_init\_\_ method for instance of *RPG*. Using a for loop that pass the number of human players to the range function, the method creates agents and stores these agents in *agent\_dict* where each can be called by *id\_num*.

1. #RPG.py
2. **import** agent
4. **class** RPG():
6. **def** \_\_init\_\_(self, num\_humans):
7. self.num\_humans= num\_humans
8. self.agent\_dict = {}
9. self.human\_ids = []
10. self.create\_players()
12. **def** create\_players(self):
13. **for** h **in** range(self.num\_humans):
14. self.agent\_dict[h] = agent.Agent(rpg = self, hp = 20, mp = 10, strength = 5, defense = 5, magic = 5, id\_num = h)
15. self.human\_ids.append(h)

We can now create an instance of the RPG and check its attributes. We will do this by creating a file named *playRPG.py*.

1. #playRPG.py
2. **import** RPG
4. rpg = RPG.RPG(num\_humans = 1)

This script will create an instance of the RPG class that will include 1 human agent. Our game will not have any battles yet, but we can check the dictionary that holds the agents.

In the console enter:

*rpg.agent\_dict*

This returns a copy of *agent\_dict*, which is owned by rpg:

*{0: <agent.Agent at 0x224fd701400>}*

The RPG class creates agents from by importing the py file *agent* and calling an instance of the *Agent* class. We can check attributes of this class. Check the *hp* of the agent created by entering:

*rpg.agent\_dict[0].hp*

This returns the hp of the agent, which is *20*.

Next we will add agents that are controlled by AI. Our AI will be the most simple possible. Agents controlled by AI in this game make random choices. They will either *attack* or *recover*. First, we need to differentiate between these *computer* agents and *human* agents. This requires editing of all 3 files to include the number of computers. We don’t want these computers to have exactly the same strength in every instance, so we will use a random number generater to create variance in attribtutes of computer players.

1. #agent.py
2. **import** random
3. **class** Agent():
5. **def** \_\_init\_\_(self, rpg, hp, mp, strength, defense, magic, wealth, id\_num, computer = True):
6. self.rpg = rpg
7. self.id\_num = id\_num
8. self.alive = True
9. self.record = {"Wins": 0, "Losses": 0}
11. **if** computer == False:
12. self.strength = strength
13. self.hp = hp
14. self.max\_hp = hp
15. self.mp = mp
16. self.max\_mp = mp
17. self.defense = defense
18. self.magic = magic
19. self.type = "human"
20. **if** computer:
21. self.strength = random.randint(int(.5 \* strength), int(strength))
22. self.hp = random.randint(int(.5 \* hp), int(hp))
23. self.max\_hp = hp
24. self.mp = random.randint(int(.5 \* mp), int(mp))
25. self.max\_mp = mp
26. self.defense = random.randint(int(.5 \* defense), int(defense))
27. self.magic = random.randint(int(.5 \* magic), int(magic))
28. self.type = "computer"

Agents are identified either as *“human”* or “*computer”*. Which type they are depends upon whether or not *computer* is *True* when passed to *Agent* in the *RPG* class.

Update the *RPG* class by adding *num\_computers*, which will identify the number of computer agents that will be created. To account for these agents, we create a list called *computer\_ids*.

We will need to edit the *create\_players* method next. In the first for loop, we create human players. This is identified by passing *computer = False* to *Agent*. Once all human agents have been created, we identify the id number of the last agent. The *id\_num* of the first computer agent will be one integer greater than that of the last human: *h + 1*. Next, create a new for loop for creating computer agents: *for c in range(lowest\_computer\_id, lowest\_computer\_id + self.num\_computers)*. Computer agents must pass the term: *computer = True*. Finally, account for which agents are computers by appending the *id\_num* of each to the *computer\_ids* list.

1. #RPG.py
2. **import** agent
4. **class** RPG():
6. **def** \_\_init\_\_(self, num\_humans, num\_computers):
7. self.num\_humans= num\_humans
8. self.num\_computers = num\_computers
9. self.agent\_dict = {}
10. self.human\_ids = []
11. self.computer\_ids = []
12. self.create\_players()
14. **def** create\_players(self):
15. **for** h **in** range(self.num\_humans):
16. self.agent\_dict[h] = agent.Agent(rpg = self, hp = 20, mp = 10, strength = 5, defense = 5, magic = 5, id\_num = h, computer = False)
17. self.human\_ids.append(h)
18. lowest\_computer\_id = h + 1
19. **for** c **in** range(lowest\_computer\_id, lowest\_computer\_id +  self.num\_computers):
20. self.agent\_dict[c] = agent.Agent(rpg = self, hp = 20, mp = 10, strength = 5, defense = 5, magic = 5, id\_num = c, computer = True)
21. self.computer\_ids.append(c)

Finally, adjust *playRPG.py* to include *num\_computers.*

1. #playRPG.py
2. **import** RPG
4. rpg = RPG.RPG(num\_humans = 1, num\_computers = 3)

Execute this script to create a game with 1 human and 3 computers. Calling *rpg.agent \_dict* will show that there are 4 agents:

{0: <agent1.Agent at 0x224fd68cbe0>,

1: <agent1.Agent at 0x224fd76e4a8>,

2: <agent1.Agent at 0x224fd76e358>,

3: <agent1.Agent at 0x224fd76e400>}

Let’s check the which of these are computers by calling *rpg.computer\_ids*:

[1, 2, 3]

Agents 1, 2, and 3 are computers.

Now we can create a battle. We will need to select

Sugarscape is an agent-based model developed for investigating social phenomena. The setup is simple. Agents occupy space on a landscape with sugar hills. Their goal is to collect from the space that has the highest value of sugar. Agents who are successful at collecting sugar are able to survive. In our version of the model, agents that are able to collect enough sugar can also reproduce. We also add a second good, water, that agents can trade for.

**Each class will exist in a file named after the class**

First, we will build a program to build the landscape. Import the text file **“name “, download it from ….**

**Create a Patch Function**

**Patches are the tiles that agents occupy. Agents move from patch to patch. We will wait to create agents. For now, we will begin building the Patch(), Model, and GUI() classes.**

1. #Patch.py
2. **class** Patch():
3. **def** \_\_init\_\_(self, model, row, col, maxQ):
4. #This links the patch to the model that creates the patch
5. self.model = model
6. self.row = row
7. self.col = col
8. #The maximum quantity of a good that patch can hold
9. self.maxQ = maxQ
10. #The current quantity of a good held by a patch
11. self.Q = maxQ

We have not yet created a model class. We will use this to have agents and patches interact across time. We will execute commands for agents and patches from here. We will be importing pandas. This will be used to import the csv file that indicates the maximum quantity of a good held on each patch. We will use this information to construct the board and create values for each Patch object.

1. **import** pandas as pd
2. **from** Patch **import** \*
3. #Model.py
4. **class** Model():
5. **def** \_\_init\_\_(self, gui):
6. #Instantiate Patches
7. sugarMap = pd.read\_csv('sugar-map.txt', header = None, sep = ' ')
8. #sugarMap.shape calls the a tuple with dimensions
9. #of the dataframe
10. self.rows, self.cols = sugarMap.shape
11. #Create a list to hold the patches. We first fill these with
12. #zeros to hold the place for each Patch object
13. self.patchList = [[0 **for** i **in** range(self.rows)] **for** j **in** range(self.cols)]
14. **for** i **in** range(self.rows):
15. **for** j **in** range(self.cols):
16. #replace zeros with actual Patch objects
17. self.patchList[i][j] = Patch(self,  i , j, sugarMap[i][j])
18. **from** tkinter **import** \*
19. **from** Model **import** \*
21. **class** GUI():
22. **def** \_\_init\_\_(self, parent):
23. self.model = Model(self)
24. self.dimPatch = 16
25. canvasWidth = self.model.cols \* self.dimPatch
26. canvasHeight= self.model.rows \* self.dimPatch
27. self.canvas = Canvas(parent, width=canvasWidth, height=canvasHeight, background="white")
28. #puts in canvas window
29. self.canvas.pack()
30. self.drawPatches()

33. **def** drawPatches(self):
34. **for** row **in** self.model.patchList:
35. **for** patch **in** row:
36. patch.image = self.canvas.create\_rectangle(
37. #left x
38. patch.col \* self.dimPatch,
39. #top y
40. patch.row \* self.dimPatch,
41. #right x
42. (patch.col + 1) \* self.dimPatch,
43. #bottom y
44. (patch.row + 1) \* self.dimPatch,
45. fill=self.color(patch.Q),#, patch.good,
46. width=0 #Border width = 0
47. )
48. #Outputs string in the format '#RRGGBB'
49. **def** color(self, q):
50. #(256 / 4) - 1  = 63
51. rgb = (255 - 3 \* q,255 - 10 \* q,255-63\*q)
52. color = '#'
53. **for** v **in** rgb:
54. # cuts off beginning of hex() output: '0x'
55. hx = hex(v)[2:]
56. **while** len(hx) < 2:
57. # add 0 to beginning if 1 characters
58. hx = '0' + hx
59. color += hx
60. **return** color
62. parent = Tk()
63. parent.title = "Sugarscape"
64. y = GUI(parent)
66. **if** \_\_name\_\_ == "\_\_main\_\_":
67. parent.mainloop()

**Extended Table of Contents**

[**Introduction 3**](#Introduction)

[**Chapter 1: The Essentials 5**](#TheEssentials)

1. **Printing**

[**Chapter 2: Working with Lists 15**](#WorkingWithLists)

[**Chapter 3: Building Functions 29**](#BuildingFunctions)

[**Chapter 4: Classes and Methods 40**](#ClassesAndMethods)

[**Chapter 5: Working With Numpy and Pandas 47**](#WorkingWithNumpyAndPandas)

[**Chapter 6: Importing, Cleaning, and Analyzing Data 67**](#ImportingCleaningAndAnalyzingData)

[**Chapter 7: Building an OLS Regression Function 74**](#BuildingAnOLSRegressionFunction)

[**Chapter 8: Agent-based Models 95**](#AgentBasedModels)