Learn Python for Economic Computation: A Crash Course

by

James Caton
Cameron Harwick

Working Draft – Do Not Distribute

Table of Contents

Introduction: Learn Python for Economic Computation

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. 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.anaconda.com/distribution/#download-section

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

Learning to program is like learning to play a new instrument. While it is useful to spend some time reading about music, your instrument, music theory, and so forth, if one wants to learn to play they must practice reading and playing music. New musicians learn to play simple phrases and songs. Think of the exercises in this book as the equivalent for programming. To begin learning, simply copy the script and execute it. You might not understand what you are doing at first. That is *okay*. Just as musicians continue to play songs that they have already "learned", you will benefit by reconstructing and further developing examples that you have already completed. Learn by doing.

a. Printing

New Concepts	Description
Strings	Data type that interprets letters and numbers as text, (not numerical values)
print()	A function that prints in the console objects passed to it
Running Script	Script must be executed. In Spyder, this can be accomplished by pressing F5.

Python offers a wide variety of tools to be used by a program. The first of these that we will learn is the *print()* method. 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.

```
#helloWorld.py
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!

Let's break down this program. You used the method *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.

```
#helloWorldSingleQuote.py
```

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

Output:

```
Hello world!
```

b. Create a String Object

New Concepts	
Variables	Objects are saved as variables. E.g., for *msg = "Hello!"*, *msg* is a variable

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*. It points to a string containing *"Hello Jim!"*

```python
#helloJim.py

msg = "Hello Jim!"
print(msg)
```

Output:

```
Hello Jim!
```

Often, it will be useful to save data as an object so that it can be called later in the script without having to be rewritten.

c. String Methods

New Concepts	Description
String Methods [i.e., str.upper()]	Methods from string objects perform an operation on the object. For example, *str.upper()* capitalizes the text saved as *str*.
String Concatenation [str1 + str2]	You may use a + to join together to separate strings as a single string. i.e., "Happy" + " Birthday" → "Happy Birthday"
Separator [*print(obj1* **,** obj2)]	Multiple object may be passed to the print function if they are separated by a comma. The comma will be interpreted by print as an object to be printed. i.e., whatever value is identied by *sep* in *print(…, sep = " ")*
Esacape Sequence ["\n"]	An escape sequence instructs print to interpret the command (e.g., "\n") as a command rather than literally interpret text
Quotes in Strings	Strings are encapsulated by quotations. To include quotations within strings, either use a different kind of

	quotation (i.e., “Use ‘single quotes’ within double quotes”) or preced the quotation mark by a backslash.

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

```python
#stringCaps.py
msg = "john nash"
print(msg)

# ALL CAPS
print(msg.upper())

# First Letter Of Each Word Capitalized
print(msg.title())
```

Output:

```
john nash
JOHN NASH
John Nash
```

We used some methods that are owned by instances of the string class in Python. The method *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.

```python
#concatenateStrings.py
# join 3 different strings and save the result as msg
msg = "john" + " " + "nash"
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.

```python
#concatenateStrings2.py
line1 = "You thought it would be easy"
line2 = "You thought it wouldn't be strange"
line3 = "But then you started coding"
line4 = "Things never were the same"
# concatenate the above strings while passing to print()
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.

```python
#concatenateStringsSpaces.py
line1 = "You thought it would be easy"
line2 = "You thought it wouldn't be strange"
line3 = "But then you started coding"
line4 = "Things never were the same"
# concatenate the above strings with spaces between them
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 function

```python
#concatenateStringsCommas.py
line1 = "You thought it would be easy"
line2 = "You thought it wouldn't be strange"
line3 = "But then you started coding"
line4 = "Things never were the same"
# using a comma between each object includes a space to separate them
print(line1, line2, line3, line4)
```

When *print()* reads a comma between objects to be printed, it automatically prints whatever character is identified by the separator. By default, this is a space: *" "*. You can change the string serving to separate each printed object by passing *sep = . . .* to *print()*. Suppose that you wanted to print each string without spaces between them. You could pass *sep = ""*.

```python
#concatenateStringsCommasNoSpaces.py
line1 = "You thought it would be easy"
line2 = "You thought it wouldn't be strange"
line3 = "But then you started coding"
line4 = "Things never were the same"
# Now the comma inserts and empty string instead of a space
print(line1, line2, line3, line4, sep = "")
```

Output:

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

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.

```python
#concatenateStringsNewLines.py
line1 = "You thought it would be easy"
line2 = "You thought it wouldn't be strange"
line3 = "But then you started coding"
line4 = "Things never were the same"
# Concatenate each string with the escape sequence that creates a new line
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
```

The same output can be generated by redefining *sep,* this time as *sep = “\n”*.

```python
 #concatenateStringsCommasNewLines.py
line1 = "You thought it would be easy"
line2 = "You thought it wouldn't be strange"
line3 = "But then you started coding"
line4 = "Things never were the same"
# Now the comma inserts a "\n" to start a new line
# instead of adding a space
print(line1, line2, line3, line4, sep = "\n")
```

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.

```python
#quotesInQuotes.py
single_in_double = "We may use 'single quotes' within double quotes"
double_in_single = 'We may use "double quotes" within single quotes'
print(single_in_double)
print(double_in_single)
```

Output:

```
We may use 'single quotes' within double quotes
We may use "double quotes" within single quotes
```

d. Escape Sequences

New Concepts	**Description**
Escape Sequences (more)	*See lists below*

Below is a list of commands including and related to the string used in earlier exercises, “\n”. These are known as 'escape sequences'.

Escape Sequence	Result
\\	Backslash
\’	Single Quote
\”	Double Quote
\t	Horizontal Tab
\n	New Line

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*.

```python
#escapeSequences.py
single_quotes = 'We may use \'single quotes\' in single quotes.'
double_quotes = "Or we may use \"double quotes\" in double quotes."
read_backslash = "We may use two backslashes to print a single backslash: \\"
lines = "Every\nword\nis\na\nnew\nline"
new_line_and_tab = "We may start a new line \n\tand use tab for a hanging indent"

print(single_quotes)
print(double_quotes)
print(read_backslash)
print(lines)
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
```

Other escape sequences include:

Escape Sequence	Result
\b	Backspace
\f	Formfeed
\n	New Line
\v	Vertical Tab[1]

See Official Documentation: https://docs.python.org/2.0/ref/strings.html

e. More String Functions

[1] The Python console in Spyder does not interpret “\v”

New Concepts	
String Methods (more) [i.e., str.lstrip(), str.rstrip(), str.strip(), str.replace()]	*str.lstrip()*: remove spaces on the far left; *str.rstrip()*: remove spaces on the far right; *str.strip()*: remove spaces on the left and right of text object; *str.replace(target, replace)*: replace instances of *target* string with *replace* string
Triple [Block] Quotes	String objects can be encapsulated by three quotation marks on either end of the string. The use of triple quotes allow for strings to include new lines without use of the backslash followed by a return.

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.

```python
#strip.py
spaces = "    Look at all the spaces in the text!   "
print("no spaces removed:\n", spaces)

remove_left_spaces = spaces.lstrip()
remove_right_spaces = spaces.rstrip()
remove_left_and_right_spaces = spaces.strip()

print("Remove left spaces:\n" + remove_left_spaces )
print("Remove right spaces:\n" + remove_right_spaces )
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.

```python
#replace.py
spaces = "    Look at all the spaces in the text!      "
print("No spaces removed:\n", spaces)

remove_all_spaces = spaces.replace(" ", "")
```

```python
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.

```python
#stringTripleQuotes.py
x = """\
Everything in this object will be recorded exactly as entered,
if we enter a new line or
    a new line with a tab."""

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

New Concepts	Description
Integers	A data type whose values are whole numbers
Floats	A data type whose values include a decimal place
Errors	Errors occur when python is unable to compute script that is executed
Typecasting	Change the type of a value (i.e., int("8") transforms a string to an integer).
Modules (sys)	Modules allow user to use programs that have already been created. They should be imported at the beginning of the script using ***import*** *module* or ***from*** *module* ***import*** *object*
Overflow Error	Overflow error occurs when a numerical value attempting to be saved exceeds the size of the value allowed by the data type.

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.

```python
#integers.py
num1 = 5 + 3
num2 = 5 - 4
num3 = 4 - 3

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.

```python
#integersVsStrings.py

num1 = 5 + 3
num1s = "5" + "3"

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.

```python
#floats.py
num1 = 5 / 3
num2 = 5 / 4
num3 = 4 / 3

print("num1:",num1, "\nnum2:", num2, "\nnum3:", num3)
```

Output:

```
num1: 1.6666666666666667
num2: 1.25
num3: 1.3333333333333333
```

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.

```python
#printError.py
num1 = 5 / 3
num2 = 5 / 4
num3 = 4 / 3
sum_nums = num1 + num2 + num3

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:

```python
#castString.py
num1 = 5 / 3
num2 = 5 / 4
num3 = 4 / 3
sum_nums = num1 + num2 + num3

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.

```python
#printMixedTypesWithComma.py
num1 = 5 / 3
num2 = 5 / 4
num3 = 4 / 3
sum_nums = num1 + num2 + num3

print("num1 + num2 + num3:", sumNums)
```

Output:

```
num1 + num2 + num3: 4.25
```

A language must indicate the amount of mermory a particular value will require 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 the *sys* library.

```python
#checkFloat.py

import sys
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 2.0^{1023} . In Python we indicate an exponent with **, as in the example below.

```python
x = 2.0 ** 1023
print(x)
```

Output:

```
8.98846567431158e+307
```

If we enter "num = 2.0 ** 1024", Python will return an overflow error, meaning that the number is beyond the capacity that can be held by the float (double) in python.

```python
x = 2.0 ** 1024
```

Output:

```
Traceback (most recent call last):

  File "<ipython-input-4-7c08c36af256>", line 1, in <module>
    2.0 **1024

OverflowError: (34, 'Result too large')
```

The problem does not occur if 2 is an integer raised to 1024 or a higher value.

```python
#checkFloat.py

import sys
print(sys.float_info)

x = 2.0 ** 1023
print(type(x))
print(x)

y = 2 ** 1025
print(type(y))
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'>
```

```
35953862697246318154586103815780494672359539578846131454686016231546535161100
19262654169546448150720422402277597427867153175795376288332449856948612789482
48755535786849730970552604439202492188238906165904170011537676301364684925762
94782622108165447432670102136917259647989449187695943260967071265924844827443
2
```

Just as with any spoken language, scripting in Python can reveal much nuance in the strcuture of the language. As we learn to program, we will come upon a number of unique problems dealing with memory usage, object types, script structure, and so forth. To the extent that we can learn about them systematically we will, otherwise we will deal with them as they appear.

Exercises

1. Record your name in an object (x = ...) and a description of yourself in another object.

2. Raise 2 to the 4th power. Save the value as an object named twoToTheFourth. Then create a new value that sums the value twice. Create another variable that saves the value of twoToTheFourth as a string. Just as with the numeric value, sum the string object twice.

3. Find the list of escape sequences in section 2.4: https://docs.python.org/3/reference/lexical_analysis.html#literals create a string that uses at least 4 different escape sequences.

4. Add an integer and a float together. Is the final number an integer or a float?

5. Add 2 ** 1024 + 1.5. What is the outcome? Why? hint: Try adding .5, 1, and 1.1 instead of 1.5.

6. import the string library with the script "import string". Print string.__dict__ . What is the output? If you do not understand the meaning, search "python class __dict__" and find an explanation.

7. Call at least 5 of the results from string.__dict__.

8. Concatenate 3 distinct strings

Chapter 2: Working With Lists

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

If you are familiar with other programming languages, you may be accustomed to working with arrays. An array is must be cast to house particular data types (float, int, string, etc…). By default, Python works with dynamic lists instead of arrays. Dynamic lists are not cast as a particular type.

a. Working with Lists

New Concepts	**Description**
Dynamic List	A dynamic list is encapsulated by brackets ([]). A list is mutable. Elements can be added to or deleted from a list on the fly.
List Concatenation	Two lists can be joined together in the same manner that strings are concatenated.
List Indexing	Lists are indexed with the first element being indexed as zero and the last element as the length of (number of elements in) the list less one. Indexes are called using brackets – i.e., lst[0] calls the 0th element in the list.

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:

```python
#lists.py

empty_list = []
int_list = [1, 2, 3, 4, 5]
float_list = [1.0, 2.0, 3.0, 4.0, 5.0]
string_list = ["Many words", "impoverished meaning"]
mixed_list = [1, 2.0, "Mix it up"]
print(empty_list)
print(int_list)
print(float_list)
print(string_list)
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:

```python
#concatenateLists.py

list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5, 3]
join_lists = list1 + list2

print("list1:", list1)
print("list2:", list2)
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.

```python
#copyListElements.py

list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5, 3]
print(“list1 elements:”, list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
print(“list2 elements:”, list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])

list3 = []
list3.append(list1[0])
list3.append(list1[1])
list3.append(list1[2])
list3.append(list1[3])
list3.append(list1[4])
list3.append(list1[5])

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()*

New Concepts	Description
list()	List transforms an iterable object, such as a tuple or set, into a dynamic list.
For Loops	A for loop calls each element of an iterable object consecutively and individually.
range(j, k, l)	Identifies a range of integers from *j* to *k – 1* separated by some interval *l.*
len()	Measure the length of an iterable object.

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 (*k* – 1), where the length of the list is the value *k*. These cases are illustrated below in *range.py*.

```python
#range.py

list1 = list(range(9))
list2 = list(range(-9,9))
list3 = list(range(-9,9,3))

print(list1)
print(list2)
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*.

```python
#forLoopAndRange.py
```

```python

for i in range(10):
    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 from 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*.

```python
#copyListElementsForLoop.py

list1 = [5,4,9,10,3,5]
list2 = [6,3,2,1,5,3]
print("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
print("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])
list3 =[]
j = len(list1)
for i in range(j):
   list3.append(list1[i])

k = len(list2)
for i in range(k):
   list3.append(list2[i])

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

New Concepts	Description
List Methods [i.e., .append(), .insert()]	List methods append and insert increse the length of a list by adding in element to the list.
If Statements	An if statement executes the block of code contained in it if conditions stipulated by the if statemenet are met (they return *True*).
Else Statement	In the case that the conditions stipulated by an if statement are not met, and else statement executes an alternate block of code
Operator [i.e., ==]	Two equals signs test the condition that the value of the variable on either side is equal.

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*.

```
#addListElements.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5, 3]
print("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
print("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])

list3 = []
j = len(list1)
for i in range(j):
    list3.append(list1[i] + list2[i])

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. 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.

```
#addListElements2.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5, 3]
print("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4])
print("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4])

list3 = []
```

```python
j = len(list1)
for i in range(j):
    list3.append(list1[i] + list2[i])
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 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 index from one list, it exists in the other. We do not want to call a list index if it does not exist. That would produce an error.

We can check if a statement is true using an if statement. 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 will want to check if the lengths of two different lists are the same. To check that a variable has a stipulated value, we use two equals signs. Using == allows the program to compare two values rather setting the value of the variable on the left, as would occur with only one equals sign.

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 at the ith element of list3.

```python
#addListElements3.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5, 3]
print("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
print("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])

list3 = []
j = len(list1)
if len(list1) == len(list2):
    for i in range(0, len(list1)):
        list3.insert(i, list1[i] + list2[i])
else:
    print("Lists are not the same length, cannot perform element-wise operations.")

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:

```
#addListElements4.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5]
print("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4])
print("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4])

list3 = []
j = len(list1)
if len(list1) == len(list2):
    for i in range(0, len(list1)):
        list3.insert(i, list1[i] + list2[i])
else:
    print("Lists are not the same length, cannot perform element-wise operations.")

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

New Concepts	Description
del	The command *del* is used to delete an element from a list

List Methods [i.e., *.pop(), .remove()*]	The method *.pop()* removes the last element of a list, allowing it to be saved as a separate object. The method *.remove()* deletes an explicitly identified element.

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.

```python
#deleteListElements.py

list1 = ["red", "blue", "orange", "black", "white", "golden"]
list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
print("lists before deletion: ")
if len(list1) == len(list2):
   for i in range(len(list1)):
        print(list1[i], "       ", list2[i])

del list1[0]
del list2[5]

print("lists after deletion: ")
if len(list1) == len(list2):
    for i in range(len(list1)):
        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"

```python
#removeListElements

list1 = ["red", "blue", "orange", "black", "white", "golden"]
list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
```

```python
print("lists before deletion: ")
if len(list1) == len(list2):
    for i in range(len(list1)):
        print(list1[i], "       ", list2[i])

list1.remove("red")
list2.remove("dog")

print("lists after deletion: ")
if len(list1) == len(list2):
    for i in range(len(list1)):
       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. 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:

```python
#removeAndSaveListElements

list1 = ["red", "blue", "orange", "black", "white", "golden"]
list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
print("lists before deletion: ")
if len(list1) == len(list2):
    for i in range(len(list1)):
        print(list1[i], "\t", list2[i])

list1_res = "red"
list2_res = "dog"
list1.remove(list1_res)
list2.remove(list2_res)

print("lists after deletion: ")
if len(list1) == len(list2):
    for i in range(len(list1)):
       print(list1[i], "       ", list2[i])

print()
```

```
21. print("Res1\tRes2”
22. 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
```

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

```
1.  #removeListElementsPop.py
2.
3.  # define list1 and list2
4.  list1 = ["red", "blue", "orange", "black", "white", "golden"]
5.  list2 = ["nose", "ice", "fire", "cat", "mouse", "dog"]
6.
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])
12.
13. #remove list elements and save them as variables "_res"
14. list1_res = list1.pop(0)
15. list2_res = list2.pop(5)
16.
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. For loops without *range()*

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.

```python
#addListElements5.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = [6, 3, 2, 1, 5, 3]
print("list1 elements:", list1[0], list1[1], list1[2], list1[3], list1[4], list1[5])
print("list2 elements:", list2[0], list2[1], list2[2], list2[3], list2[4], list2[5])

list3 = []
j = len(list1)
if j == len(list2):
    for i in range(0, j, 2):
        list3.append(list1[i] + list2[i])
else:
    print("Lists are not the same length, cannot perform element-wise operations.")
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].

For loops in python can call in sequence element of objects that are iterable. These include lists, strings, keys and values from dictionaries, as well as the range function we have already used. 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*.

```python
#forLoopWithoutIndexer.py
list1 = ["red", "blue", "orange", "black", "white", "golden"]
list2 = []
for x in list1:
    list2.append(x)

print("list1", "\tlist2")

if len(list1) == len(list2):
   for i in range(0, len(list1)):
        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

New Concepts	
sorted()	The function *sorted()* sorts a list in order of numerical or alphabetical value.
passing errors [i.e., *try* and *except*]	A try statement will pass over an error if one is generated by the code in the *try* block. In the case that an error is passed, code from the *except* block well be called. This should typically identify the type of error that was passed.

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.

```python
#sorting.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = ["red", "blue", "orange", "black", "white", "golden"]

```

```python
print("list1:", list1)
print("list2:", list2)

sorted_list1 = sorted(list1)
sorted_list2 = sorted(list2)

print("sorted_list1:", sorted_list1)
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:

```python
#sortingError.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = ["red", "blue", "orange", "black", "white", "golden"]
list3 = list1 + list2

print("list1:", list1)
print("list2:", list2)
print("list3:", list3)

sortedList1 = sorted(list1)
sortedList2 = sorted(list2)
sortedList3 = sorted(list3)

print("sortedList1:", sortedList1)
print("sortedList2:", sortedList2)
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:

```python
#sortingError.py
list1 = [5, 4, 9, 10, 3, 5]
list2 = ["red", "blue", "orange", "black", "white", "golden"]
list3 = list1 + list2

print("list1:", list1)
print("list2:", list2)
print("list3:", list3)

```

```
sortedList1 = sorted(list1)
sortedList2 = sorted(list2)

print("sortedList1:", sortedList1)
print("sortedList2:", sortedList2)
try:
    sortedList3 = sorted(list3)
    print("sortedList3:", sortedList3)
except:
    print("TypeError: unorderable types: str() < int()"
     "Ignoring error")

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']
TypeError: unorderable types: str() < int()Ignoring error
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

New Concepts	Description
Slice *[i.e., lst[a:b]]*	A slice of a list is a copy of a portion (or all) of a list from index *a* to *b – 1*.

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:

```
#fullSlice.py
some_list = [3,1,5,6,1]
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):

```
#fullSlice2.py
some_list = [3, 1, 5, 6, 1]
```

```python
min_index = 0
max_index = len(some_list)
print("minimum:", min_index)
print("maximum:", max_index)
print("Full list using slice", some_list[min_index:max_index])
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:

```python
#partialSlice.py
min_index = 3
max_index = 7
full_list = [1,2,3,4,5,6,7,8,9]
partial_list = fullList[min_index:max_index]
print(full_list)
print(partial_list)

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. Nested For Loops

New Concepts	Description
Nested For Loops	A for loop may contain other for loops; useful for multidimensional data structures.

Creative use of for loops can save the programmer a lot of work. While you should be careful not to create so many layers of for loops and if statements that code is difficult to interpret ("Flat is better than nested"), you should be comfortable with the structure of nested for loops and, eventually, their use in structures like dictionaries and generators.

A useful way to become acquainted with the power of multiple for loops is to identify what each the result of each iteration of nested for loops. In the code below, the first for loop will count from 0 to 4. For each value of i, the second for loop will cycle through values 0 to 4 for j.

```python
#nestedForLoop.py
```

```python
print("i","j")
for i in range(5):
    for j in range(5):
        print(i,j)
```

Output:

```
i  j
0 0
0 1
0 2
0 3
0 4
1 0
1 1
1 2
1 3
1 4
2 0
2 1
2 2
2 3
2 4
3 0
3 1
3 2
3 3
3 4
4 0
4 1
4 2
4 3
4 4
```

Often, we will want to employ values generated by for loops in a manner other than printing the values generated directly by the for loops. We may, for example, want to create a new value constructed from *i* and *j*. Below, this value is constructed as the sum of *i* and *j*.

```python
#nestedForLoop.py
print("i","j", "i+j")
for i in range(5):
    for j in range(5):
        val = i + j
        print(i,j, val)
```

Output:

```
i  j i+j
0 0 0
0 1 1
```

```
0 2 2
0 3 3
0 4 4
1 0 1
1 1 2
1 2 3
1 3 4
1 4 5
2 0 2
2 1 3
2 2 4
2 3 5
2 4 6
3 0 3
3 1 4
3 2 5
3 3 6
3 4 7
4 0 4
4 1 5
4 2 6
4 3 7
4 4 8
```

If we interpret the results as a table, we can better understand the intuition of for loops. Lighter shading indicates lower values of *i* with shading growing darker as the value of *i* increases.

		j				
		0	1	2	3	4
i	0	0	1	2	3	4
	1	1	2	3	4	5
	2	2	3	4	5	6
	3	3	4	5	6	7
	4	4	5	6	7	8

i. Lists, Lists, and More Lists

New Concepts	**Description**

min(lst)
max(lst)
generators [i.e., *[val for val in lst]*]
	The function *min()* returns the lowest value from a list of values.
	The function *max()* returns that highest value from a list of values.
	Generators use a nested for loop to create an iterated data structure.

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

```
#minMaxFunctions.py
```

```python

list1 = [20,30,40,50]
max_list_value = max(list1)
min_list_value = min(list1)
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:

```python
#minMaxFunctionsByHand.py

list1 = [20, 30, 40, 50]

### initial smallest value is very high
### will be replaced if a value from the list is lower
min_list_value = 2 ** 1023

### initial largest value is very low
### will be replaced if a value from the list is higher
max_list_value = -2 ** 1023

for x in list1:
    if x < min_list_value:
        min_list_value = x
    if x > max_list_value:
        max_list_value = x

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.

```python
#listFromGenerator.py

generator = (i for i in range(20))
print(generator)

list1 = list(generator)
print(list1)

```

```python
list2 = [2 * i for i in range(20)]
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]
```

Exercises

1. Create a list of numbers 100 elements in length that counts by 3s - i.e., [3,6,9,12,...]

2. Using the list from question 1, create a second list whose elements are the same values converted to strings. hint: use a for loop and the function *str()*.

3 Using the list from question 2, create a variable that concatenates each of the elements in order of index (Hint: result should be like "36912...").

4 Using .pop() and .append(), create a list whose values are the same as the list from question 1 but in reverse order. (Hint: .pop() removes the last element from a list. The value can be save, i.e., x = lst.pop().)

5 Using len(), calculate the midpoint of the list from question 1. Pass this midpoint to slice the list so that the resultant copy includes only the second half of the list from question 1.

6 Create a string that includes only every other element, starting from the 0th element, from the string in question 3 while maintaining the order of these elements (Hint: this can be done be using a for loop whose values are descending).

7. Explain the difference between a dynamic list in Python (usually referred to as a list) and a tuple.

Exploration:

1. Use a generator to create a list of the first 100 prime numbers. Include a paragraph explaining how a generator works.

2. Using a for loop and the pop function, create a list of the values from the list of prime numbers whose values are descending from highest to lowest.

3. Using either of prime numbers, create another list that includes the same numbers but is randomly ordered. Do this without shuffling the initial list (Hint: you will need to import random and import copy). Explain in a paragraph the process that you followed to accomplish the task.

Chapter 3: Building Functions

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

New Concepts	**Description**
Building Functions	A function is a generalized block of script. For example, a function name function is declared *def function(obj1, obj2, . . .)* and defines how objects passed to the function will be handled in the function.
return obj from function	Functions may return a value to be saved if a variable is properly placed in the script such that var1 = *function(obj1, obj2, . . .)*

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

```
def function_name(object1, object2, ...):
    <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:

$$\sum_{i=0}^{n-1} x_i$$

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:

```
for values in range(n):
    <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:

```
#statsFunction1.py
```

```
def total(list_obj):
    # create variable total; value should
    # be 0 since each number in the list
    # is added to total
    total_ = 0
    # n is length of object
    n = len(list_obj)
    # n used to call each element in list using for loop
    for i in range(n):
        # add value in list to total
        total_ += list_obj[i]
    #return the sum of values in list to be saved as variable
    return total_

list1 = [3, 6, 9, 12, 15]
list2 = [i ** 2 for i in range(3,9)]
total_list1 = total(list1)
total_list2 = total(list2)
print("total_list1:", total_list1)
print("total_list2:", total_list2)
```

Output:

```
total_list1: 45
total_list2: 199
```

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

New Concepts	**Description**
Operators [i.e., !=, % , +=, **]	The operater *!=* tests whether the values on either side of the operator are equal; *a % b* returns the remainder of *a / b*; *a += b* sets a equal to *a + b*; *a ** b* raises *a* to the *b* power (a^b).
Dictionary	A dictionary is a datastructure that uses keys instead of index values. Each unique key references an object linked to that key.
Dictionary Methods [i.e., dct.values()	dct.values() returns a list of the objects that are referenced by the dictionaries keys.
Default Function Values	Function may assume a default value for values passed to it. (i.e., *def function(val1 = 0, val2 = 2, …)*)

i. Average Statistics

We define the mean of a set of numbers:

$$\frac{\sum_{i=0}^{n-1} x_i}{n}$$

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

```python
#statsFunction2.py
. . .

def mean(list_obj):
    n = len(list_obj)
    mean_ = total(list_obj) / n
    return mean_

. . .
mean_list1 = mean(list1)
mean_list2 = mean(list2)
print("mean_list1:", mean_list1)
print("mean_list2:", mean_list2)
```

Output:

```
total_list1: 45
total_list2: 199
mean_list1: 9.0
mean_list2: 33.166666666666664
```

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.

```python
#statsFunction3.py
```

```python
. . .
def median(list_obj):
    n = len(list_obj)
    list_obj = sorted(list_obj)
    #l ists of even length divided by 2 have remainder 0
    if n % 2 != 0:
        #list length is odd
        middle_index = int((n - 1) / 2)
        median_ = list_obj[middle_index]
    else:
        upper_middle_index = int(n/2)
        lower_middle_index = upper_middle_index - 1
        # pass slice with two middle values to mean()
        median_ = mean(list_obj[lower_middle_index:upper_middle_index + 1])

    return median

. . .
median_list1 = median(list1)
median_list2 = median(list2)
print("median_list1:", median_list1)
print("median_list2:", median_list2)
```

Output:

```
total_list1: 45
total_list2: 199
mean_list1: 9.0
mean_list2: 33.166666666666664
median_list1: 9
median_list2: 30.5
```

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.

```python
#statsFunction4.py
...
def mode(list_obj):
    # use to record value[s] that appear most times
    max_count = 0
    # use to count occurrences of each value in list
    counter_dict = {}
    for value in list_obj:
        # count for each value should start at 0
        counter_dict[value] = 0
    for value in list_obj:
        # add one to the count for of the value for each occurence in list_obj
        counter_dict[value] +=1
    #make a list of the value (not keys) from the dictionary. . .
    count_list = list(counter_dict.values())
    # and find the max value
    max_count = max(count_list)
    # use a generator to make a list of the values (keys) whose number of
    # occurrences in the list match max_count
    mode_ =[key for key in counter_dict if counter_dict[key] == max_count]

    return mode_
. . .
mode_list1 = mode(list1)
mode_list2 = mode(list2)
print("mode_list1:", mode_list1)
print("mode_list2:", mode_list2)
```

Output:

```
total_list1: 45
total_list2: 199
mean_list1: 9.0
mean_list2: 33.166666666666664
median_list1: 9
median_list2: 30.5
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49, 64]
```

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

$$var_{pop} = \frac{\sum_{i=0}^{j}(x_i - x_{mean})^2}{n}$$

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

$$var_{samp} = \frac{\sum_{i=0}^{j}(x_i - x_{mean})^2}{n-1}$$

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

```python
1.  #statsFunction5.py
2.  . . .
57.
58. def variance(list_obj, sample = False):
59.     # popvar(list) = sum((xi - list_mean)**2) / n for all xi in list
60.     # save mean value of list
61.     list_mean = mean(list_obj)
62.     # use n to calculate average of sum squared diffs
63.     n = len(list_obj)
64.     # create value we can add squared diffs to
65.     sum_sq_diff = 0
66.     for val in list_obj:
67.         # adds each squared diff to sum_sq_diff
68.         sum_sq_diff += (val - list_mean) ** 2
69.     if sample == False:
70.         # normalize result by dividing by n
71.         variance_ = sum_sq_diff / n
72.     else:
73.         # for samples, normalize by dividing by (n - 1)
74.         variance_ = sum_sq_diff / (n - 1)
75.
76.     return variance_
77. . . .
96. variance_list1 = variance(list1)
97. variance_list2 = variance(list2)
98. print("variance_list1", variance_list1)
99. print("variance_list2", variance_list2)
```

Output:

```
total_list1: 45
total_list2: 135
mean_list1: 9.0
mean_list2: 27.0
median_list1: 9
median_list2: 25
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49]
variance_list1 18.0
variance_list2 202.8
```

We include an option to identify sample variance in lines 72 through 74. We can call the sample variance by adding the following in the script:

```python
100. sample_variance_list1 = variance(list1, sample = True)
101. sample_variance_list2 = variance(list2, sample = True)
102. print("sample_variance_list1:", sample_variance_list1)
103. print("sample_variance_list2:", sample_variance_list2)
```

Output:

```
. . .
sample_variance_list1: 22.5
sample_variance_list2: 430.9666666666667
```

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

$$SD = \sqrt{var}$$

This is true for both the population and sample standard deviations. The function and its employment are listed below:

```python
#statsFunction6.py
. . .

def SD(list_obj, sample = False):
    # Standard deviation is the squareroot of variance
    SD_ = variance(list_obj, sample) ** (1/2)

    return SD_
. . .
SD_list1 = SD(list1)
SD_list2 = SD(list2)
print("SD_list1:", SD_list1)
print("SD_list2:", SD_list2)
sample_SD_list1 = SD(list1, sample = True)
sample_SD_list2 = SD(list2, sample = True)
print("sample_SD_list1:", sample_SD_list1)
print("sample_SD_list2:", sample_SD_list2)
```

Output:

```
total_list1: 45
total_list2: 199
mean_list1: 9.0
mean_list2: 33.166666666666664
median_list1: 9
median_list2: 30.5
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49, 64]
variance_list1 18.0
variance_list2 359.1388888888889
sample_variance_list1: 22.5
sample_variance_list2: 430.9666666666667
SD_list1: 4.242640687119285
SD_list2: 18.950960104672504
sample_SD_list1: 4.743416490252569
sample_SD_list2: 20.75973667141919
```

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:

$$cov_{pop,\ xy} = \frac{\sum_{i=0}^{j}(x_i - x_{mean})(y_i - y_{mean})}{n}$$

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

$$cov_{pop,\ xy} = \frac{\sum_{i=0}^{j}(x_i - x_{mean})(y_i - y_{mean})}{n-1}$$

In order for covariance to be calculated, it is required that the lists passed to the function are of equal length. Thus, the function includes an *else* statementthat

```python
#statsFunction7.py
. . .
def covariance(list_obj1, list_obj2, sample = False):
    # determine the mean of each list
    mean1 = mean(list_obj1)
    mean2 = mean(list_obj2)
    # instantiate a variable holding the value of 0; this will be used to
    # sum the values generated in the for loop below
    cov = 0
    n1 = len(list_obj1)
    n2 = len(list_obj2)
    # check list lengths are equal
    if n1 == n2:
        # sum the product of the diferrence between each observation and the
        # mean for var1 and var2
        for i in range(n1):
            cov += (list_obj1[i] - mean1) * (list_obj2[i] - mean2)
        if sample == False:
            cov = cov / n1
        # account for sample by dividing by one less than number of elements
        # in list
        else:
            cov = cov / (n1 - 1)
        #return covariance
        return cov
    else:
        print("List lengths are not equal")
        print("List1 observations:", n1)
        print("List2 observations:", n2)
. . .
population_cov= covariance(list1, list2)
print("population_cov:", population_cov)
sample_cov = covariance(list1, list2, sample = True)
```

```python
print("sample_cov:", sample_cov)
```

If you execute this code, the console will identify that the lengths of the list are not the same.

Output:

```
total_list1: 45
total_list2: 199
mean_list1: 9.0
mean_list2: 33.166666666666664
median_list1: 9
median_list2: 30.5
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49, 64]
variance_list1 18.0
variance_list2 359.1388888888889
sample_variance_list1: 22.5
sample_variance_list2: 430.9666666666667
SD_list1: 4.242640687119285
SD_list2: 18.950960104672504
sample_SD_list1: 4.743416490252569
sample_SD_list2: 20.75973667141919
List lengths are not equal
List1 observations: 5
List2 observations: 6
population_cov: None
List lengths are not equal
List1 observations: 5
List2 observations: 6
sample_cov: None
```

To fix this, let’s rewrite *list2* with one less *1*. Then, it will have 5 elements instead of 6.

```python
    list1 = [3, 6, 9, 12, 15]
    list2 = [i ** 2 for i in range(3,8)]
```

Output:

```
total_list1: 45
total_list2: 135
mean_list1: 9.0
mean_list2: 27.0
median_list1: 9
median_list2: 25
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49]
variance_list1 18.0
```

```
variance_list2 202.8
sample_variance_list1: 22.5
sample_variance_list2: 253.5
SD_list1: 4.242640687119285
SD_list2: 14.24078649513432
sample_SD_list1: 4.743416490252569
sample_SD_list2: 15.921683328090658
population_cov: 60.0
sample_cov: 75.0
```

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

$$corr_{pop,\ xy} = \frac{cov_{pop,\ xy}}{\sigma_x \sigma_y}$$

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.

```
#statsFunction8.py
. . .

    def correlation(list_obj1, list_obj2):
        # corrxy = cov(x,y) / (SD(X) * SD(y))
        cov = covariance(list_obj1, list_obj2)
        SD1 = SD(list_obj1)
        SD2 = SD(list_obj2)
        corr = cov / (SD1 * SD2)
        return corr
    . . .
    corr_1_2 = correlation(list1, list2)
    print("corr_1_2:", corr_1_2)
```

Output:

```
45
Mean of list1: 9.0
Median of list1: 9
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: 2.2222222222222222
Standard deviation of list1: 4.242640687119285
Standard deviation of list2: 1.4907119849998596
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: 0.7071067811865477
```

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.

$$skew_{pop} = \frac{\sum_{i=0}^{j}(x_i - x_{mean})^3}{n\sigma^3}$$

$$skew_{samp} = \frac{n\sum_{i=0}^{j}(x_i - x_{mean})^3}{(n-1)(n-2)\sigma^3}$$

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.

```python
#statsFunction9.py
. . .
def skewness(list_obj, sample = False):
    mean_ = mean(list_obj)
    skew = 0
    n = len(list_obj)
    for val in list_obj:
        skew += (val - mean_) ** 3
    skew = skew / n if not sample else n * skew / ((n - 1) * (n - 2))
    SD_ = SD(list_obj, sample)
    skew = skew / (SD_ ** 3)

    return skew
. . .
population_skew_list1 = skewness(list1,sample = False)
population_skew_list2 = skewness(list2,sample = False)
print("population_skew_list1:", population_skew_list1)
print("population_skew_list2:", population_skew_list2)
sample_skew_list1 = skewness(list1, sample = True)
sample_skew_list2 = skewness(list2, sample = True)
print("sample_skew_list1:", sample_skew_list1)
print("sample_skew_list2:", sample_skew_list2)
```

Output:

```
total_list1: 45
total_list2: 135
mean_list1: 9.0
mean_list2: 27.0
median_list1: 9
median_list2: 25
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49]
variance_list1 18.0
variance_list2 202.8
sample_variance_list1: 22.5
sample_variance_list2: 253.5
SD_list1: 4.242640687119285
SD_list2: 14.24078649513432
sample_SD_list1: 4.743416490252569
sample_SD_list2: 15.921683328090658
population_cov: 60.0
sample_cov: 75.0
corr_1_2: 0.9930726528736967
population_skew_list1: 0.0
population_skew_list2: 0.2912710611524211
sample_skew_list1: 0.0
sample_skew_list2: 0.43420126174354107
```

Kurtosis is an absolute measure of the weight of outliers. While 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.

$$kurt_{pop} = \frac{\sum_{i=0}^{j}(x_i - x_{mean})^4}{n\sigma^4}$$

$$kurt_{samp} = \frac{n(n+1)\sum_{i=0}^{j}(x_i - x_{mean})^4}{(n-1)(n-2)(n-3)\sigma^4} - \frac{3(n-1)^2}{(n-2)(n-3)}$$

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.

```python
#statsFunction10.py
. . .
def kurtosis(list_obj, sample = False):
    mean_ = mean(list_obj)
    kurt = 0
    n = len(list_obj)
    for x in list_obj:
        kurt += (x - mean_) ** 4
    SD_ = SD(list_obj, sample)
    kurt = kurt / (n * SD_ ** 4) if not sample else \
        n * (n + 1) * kurt / ((n - 1) * (n - 2) * (SD_ ** 4)) -\
        (3 *(n - 1) ** 2) / ((n - 2) * (n - 3))

    return kurt
. . .
population_kurt_list1 = kurtosis(list1)
population_kurt_list2 = kurtosis(list2)
print("population_kurt_list1:", population_kurt_list1)
print("population_kurt_list2:", population_kurt_list2)
sample_kurt_list1 = kurtosis(list1, sample = True)
sample_kurt_list2 = kurtosis(list2, sample = True)
print("sample_kurt_list1:", sample_kurt_list1)
print("sample_kurt_list2:", sample_kurt_list2)
```

Output:

```
total_list1: 45
total_list2: 135
mean_list1: 9.0
mean_list2: 27.0
median_list1: 9
median_list2: 25
mode_list1: [3, 6, 9, 12, 15]
mode_list2: [9, 16, 25, 36, 49]
variance_list1 18.0
variance_list2 202.8
sample_variance_list1: 22.5
sample_variance_list2: 253.5
SD_list1: 4.242640687119285
SD_list2: 14.24078649513432
sample_SD_list1: 4.743416490252569
sample_SD_list2: 15.921683328090658
population_cov: 60.0
sample_cov: 75.0
corr_1_2: 0.9930726528736967
population_skew_list1: 0.0
population_skew_list2: 0.2912710611524211
sample_skew_list1: 0.0
sample_skew_list2: 0.43420126174354107
population_kurt_list1: 1.7000000000000004
population_kurt_list2: 1.7528272819579145
sample_kurt_list1: 5.6
sample_kurt_list2: 6.022618255663312
```

C. Using a Nested Dictionary to Organize Statistics

New Concepts	Description
dct.keys()	This comman calls keys associated with the dictionary.
Using a dictionary in a for loop	When a dictionary is called in a for loop in the form, for key in dct, the for loop will iterate through dct.keys().

Using a dictionary, we can cleanly organizes the statistics that we have generated. Create a new script that includes all of the functions that we created in the previous lesson. We will use the same two lists that we previously created.

Next create a dictionary named *stats_dict* that will hold not only these lists, but also statistics associated with the lists. At the top level, the dictionary will have two keys: 1 and 2, referring to list1 and list2, respectively. In the lext layer, we will first save the appropriate list identified by each top layer key (i.e., 1 or 2) under the second layer key of *"list"*.

```
#statsDict.py
. . .

list1 = [3, 6, 9, 12, 15]
list2 = [i ** 2 for i in range(3,8)]
### Build a nested dictionary with lists ###
stats_dict = {}
# 1 refers to list1 and attributes associated with it
stats_dict[1] = {}
stats_dict[1]["list"] = list1
# 2 refers to list2 and attributes associated with it
stats_dict[2] = {}
stats_dict[2]["list"] = list2
```

Now that the dictionary has been created, the keys of the stats_dict by typing the following command in the console:

```
stats_dict.keys()
```

This yield:

```
dict_keys([1, 2])
```

We see that *stats_dict* has two keys: *1* and *2*. These each have been linked to their own dictionaries, thus creating a dictionary of dictionaries. Next, we will use a for loop to call these keys and create entries of each appropriate statistic (population) for the lists saved in the dictionary.

```

# for loop will call keys from stats_dict (i.e., first 1, and then 2)
for key in stats_dict:
    # save the list associated with key as lst; this will be easier to access
    lst = stats_dict[key]["list"]
    # use the functions to calculate each statistic and save in stats_dict[key]
    stats_dict[key]["total"] = total(lst)
    stats_dict[key]["mean"] = mean(lst)
    stats_dict[key]["median"] = median(lst)
    stats_dict[key]["mode"] = mode(lst)
    stats_dict[key]["variance"] = variance(lst)
    stats_dict[key]["standard deviation"] = SD(lst)
    stats_dict[key]["skewness"] = skewness(lst)
    stats_dict[key]["kurtosis"] = kurtosis(lst)

print(stats_dict)
```

Output:

```
{1: {'list': [3, 6, 9, 12, 15], 'total': 45, 'mean': 9.0, 'median': 9, 'mode': [3, 6, 9, 12, 15], 'variance':
18.0, 'standard deviation': 4.242640687119285, 'skewness': 0.0, 'kurtosis':
1.7000000000000004}, 2: {'list': [9, 16, 25, 36, 49], 'total': 135, 'mean': 27.0, 'median': 25,
'mode': [9, 16, 25, 36, 49], 'variance': 202.8, 'standard deviation': 14.24078649513432,
'skewness': 0.2912710611524211, 'kurtosis': 1.7528272819579145}}
```

Exercises

1. Create a list of random numbers between 0 and 100 whose length is 1000. (Hint: import random; search "python random" to learn more about the library.)

2. Use the variance function from the textbook to find the variance of this list. Assume that the list represent a population in whole.

3. Create a 9 more lists of the same length whose elements are random numbers between 0 and 100. Use a nested dictionary to house and identify these lists. Keys for the first layer should be the numbers 1 through 10. Lists should be stored using a second key as follows: dict_name[index]["list"]. Index represents the particular integer key between 1 and 10 as noted above.

4. Find the variance of each list and store it as follows: dict_name[index]["variance"].

5. At the end of chapter 2, we used for loops to find min and max values. Create a min() function and max() function and pass the values from the list in question 1 to each of these to determine the min and max values in that list.

6. Explain why it might be advantageous to create a function instead of building all commands from scratch as you create a script.

Exploration:

1. Visit the Python Essentials lesson from Sargent and Stachurski. Complete exercise 3. Pass 3 other sentences to the function that you create. Include a paragraph that explains in detail how the function operates

2. Visit the Python Essentials lesson from Sargent and Stachurski. Complete exercise 4. Pass 3 pairs of unique lists to the function. Include a paragraph explaining in detail how the function operates, including explanation for a solution that uses set().

Chapter 4: Classes and Methods

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

New Concepts	Description
Class	Classes are *the* fundamental element of object oriented programming. Classes provide a template that defines instances of the class. Objects that are instances of a class share attributes defined by the constructor, in addition to other attributes they may share.
function(. . . , *args)	Passing *args to a function treats the passed arguments as a tuple and performs a specified operation upon the tuple's elements.

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.

```
class Arithmetic():
    def __init__(self):
        pass
```

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

```

arithmetic = Arithmetic()
print(arithmetic)
```

Output:

```
<__main__.Arithmetic object at 0x0000020E720A5A20>
```

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.

```python
class Arithmetic():
    def __init__(self):
        pass

    def add(self, *args):
        try:
            total = 0
            for arg in args:
                total += arg
            return total

        except:
            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

```python
arithmetic = Arithmetic()
print(arithmetic.target_value)
print(arithmetic.add(1,2,3,4,5,6,7,8,9,10))
```

Output:

```
0
55
```

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.

```python
#arithmetic.py
class Arithmetic():

    def __init__(self,  ):
        pass

    def add(self, *args):
        try:
            total = sum([arg for arg in args])
            return total
        except:
            print("Pass only int or float to add()")

    def multiply(self, *args):
        product = 1
        try:
            for arg in args:
                product *= arg
            return product
        except:
            print("Pass only 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.

```

arithmetic = Arithmetic()
print(arithmetic)
print(arithmetic.add(1,2,3,4,5,6,7,8,9,10))
print(arithmetic.multiply(2,3,4))
```

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.

Output:

```
0
55
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, a, where $a = Base^{exponent}$.

```
#arithmetic.py
class Arithmetic():

    def __init__(self,  ):
        pass

    def add(self, *args):
        try:
            total = sum([arg for arg in args])
            return total
        except:
            print("Pass only int or float to add()")

    def multiply(self, *args):
        product = 1
        try:
            for arg in args:
```

```python
                product *= arg
            return product
        except:
            print("Pass only int or float to multiply()")

    def power(self, base, exponent):
        try:
            value = base ** exponent
            return value
        except:
            print ("Pass int or float for base and exponent")

arithmetic = Arithmetic()
print(arithmetic)
print(arithmetic.add(1,2,3,4,5,6,7,8,9,10))
print(arithmetic.multiply(2,3,4))
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)".

After creating *stats.py* with the *Stats* class, we will import stats using another python script in the same folder.

```python
#stats.py
class Stats():
    def __init__(self):
        print("You created an instance of Stats")
    def total(self, list_obj):
        total = 0
        n = len(list_obj)
        for i in range(n):
            total += list_obj[i]

        return total

    def mean(self, list_obj):
        n = len(list_obj)
        mean = self.total(list_obj) / n
        return mean

    def median(self, list_obj):
        n = len(list_obj)
        list_obj = sorted(list_obj)
        if n % 2 != 0:
            middle_index = int((n - 2) / 2)
            median = list_obj[middle_index]
        else:
            upper_middle_index = int(n / 2)
            lower_middle_index = upper_middle_index - 1
            median = self.mean(\
                list_obj[lower_middle_index:upper_middle_index + 1])

        return median

    def mode(self, list_obj):
        max_count = 0
        counter_dict={}
        for value in list_obj:
            counter_dict[value] = 0
        for value in list_obj:
            counter_dict[value] += 1
        count_list = list(counter_dict.values())
        max_count = max(count_list)
        mode = [key for key in counter_dict if counter_dict[key] == max_count]

        return mode

    def variance(self, list_obj, sample = False):
        list_mean = self.mean(list_obj)
        n = len(list_obj)
        sum_sq_diff = 0
        for val in list_obj:
            sum_sq_diff += (val - list_mean) ** 2
        if sample == False:
            variance = sum_sq_diff / n
        else:
            variance = sum_sq_diff / (n - 1)

        return variance

    def SD(self, list_obj, sample = False):
        SD = self.variance(list_obj, sample) ** (1/2)
        return SD

    def covariance(self, list_obj1, list_obj2, sample = False):
        mean1 = self.mean(list_obj1)
        mean2 = self.mean(list_obj2)
        cov = 0
        n1 = len(list_obj1)
        n2 = len(list_obj2)

        if n1 == n2:
            for i in range(n1):
                cov += (list_obj1[i] - mean1) * \
                    (list_obj2[i] - mean2)
            if sample == False:
                cov = cov / n1
            else:
                cov = cov / (n1 - 1)
            return cov
        else:
            print("List lengths are not equal")
            print("List1 observations:", n1)
            print("List2 observations:", n2)

    def correlation(self, list_obj1, list_obj2):
        cov = self.covariance(list_obj1, list_obj2)
        SD1 = self.SD(list_obj1)
        SD2 = self.SD(list_obj2)
        corr = cov / (SD1 * SD2)

        return corr

    def skewness(self, list_obj, sample = False):
        mean_ = self.mean(list_obj)
        skew = 0
        n = len(list_obj)
        for val in list_obj:
            skew += (val - mean_) ** 3
        skew = skew / n if not sample else n * skew / ((n-1) * (n - 2))
        SD_ = self.SD(list_obj, sample)
        skew = skew / (SD_ ** 3)

        return skew

    def kurtosis(self, list_obj, sample = False):
        mean_ = self.mean(list_obj)
        kurt = 0
        n = len(list_obj)
        for x in list_obj:
            kurt += (x - mean_) ** 4
        SD_ = self.SD(list_obj, sample)
        kurt = kurt / (n * SD_ ** 4) if not sample else \
            n * (n + 1) * kurt / ((n - 1) * (n - 2) * (SD_ ** 4)) -\
            (3 *(n - 1) ** 2) / ((n - 2) * (n - 3))

        return kurt
```

We will import *stats.py* using a separate script called *importStats.py*. Once this script is imported, call the class with stats and name the instance of *Stats*, *stats_lib*.

```python
import stats

stats_lib = stats.Stats()
```

Below, we use each method in *stats_lib* to check that we correctly transformed the functions into methods of the class *Stats*.

```python
list1 = [3, 6, 9, 12, 15]
list2 = [i ** 2 for i in range(3,8)]
print("sum list1 and list2", stats_lib.total(list1 + list2))
print("mean list1 and list2", stats_lib.mean(list1 + list2))
print("median list1 and list2", stats_lib.median(list1 + list2))
print("mode of list1 and list2", stats_lib.mode(list1 + list2))
print("variance of list1 and list2", stats_lib.variance(list1 + list2))
print("standard deviation of list1 and list2",
      stats_lib.SD(list1 + list2))
print("covariance of list1 and list2 (separate)",
      stats_lib.covariance(list1, list2))
print("correlation of list1 and list2 (separate)",
      stats_lib.correlation(list1, list2))
print("skewness of list1 and list2", stats_lib.skewness(list1 + list2))
print("kurtosis of list1 and list2", stats_lib.kurtosis(list1 + list2))
```

Output:

```
You created an instance of Stats
sum list1 and list2 180
mean list1 and list2 18.0
median list1 and list2 13.5
mode of list1 and list2 [9]
variance of list1 and list2 191.4
standard deviation of list1 and list2 13.83473888441701
covariance of list1 and list2 (separate) 60.0
correlation of list1 and list2 (separate) 0.9930726528736967
skewness of list1 and list2 1.1009947530550708
kurtosis of list1 and list2 3.048466504849597
```

Exercises

1. Create a function that calculates the length of a list without using len() and returns this value. Create a list and pass it to the function to find its length.

2. Create a function that performs dot multiplication on two vectors (lists) – such that if list1 = [x1,x2,x3] and list2 = [y1,y2,y3], dot_product_list1_list2 = [x1y1, x2y2, x3y3] – and returns this list. Pass two lists of the same length to this function.

3. In a single line, pass two lists of the same length to the function from question 2 and pass the instance of that function to the function from question 1. What is the length of dot_product_list1_list2?

4. Create two unique lists using generator functions and pass them to the function created in question 2.

5. Create a function that checks the types of elements in a list. For example, if a list contains a string, an integer, and a float, this function should return a list that contains identifies these three types: [str, int, float].

6. In a single line, pass a list with at least 4 different types to the function from question 5 and pass the result to the funciton measuring length.

7. Create a class that houses each of the functions (now methods) that you have created. Create an instance of that class and use each of the methods from the class.

Exploration

1. Visit OOP II: Building Classes lesson from Sargent and Stachurski and duplicate "Example: A Consumer Class". Following this, pass different values to the class methods and return the value of agent wealth using object.__dict__ write a paragraph explaining the script and the results.

2. Visit OOP II: Building Classes lesson from Sargent and Stachurski and duplicate "Example: The Solow Growth Model". Following this, pass different values for each of the parameters and show how the output changes. Write a paragraph explaining the script and your findings.

5. Introduction to *numpy, pandas, and matplotlib*

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 dynamic lists. Pandas, which uses *numpy*, helps to organize data in tables. Dataframes from *pandas* can also be used to make data visualization in *matplotlib* a breeze.

a. *numpy*

New Concepts	Description
numpy	The numpy package provides data structures including arrays and linear algebra matrices, as well as efficient calculation processes.
numpy methods	*np.array(), np.arange(), np.zeros(), np.ones(), np.zeros_like(), np.ones_like(), np.empty_like()*
two-dimensional lists and arrays	Lists that hold other lists and arrays that hold other arrays are multi-dimensional. We will work with two dimensional arrays in this section.

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*.

```python
#numpyArray.py
import numpy as np

array = np.array([1, 2, 3, 4, 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

```python
#numpyArray.py
import numpy as np

array = np.array([1, 2, 3, 4, 5])
print(array)
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.

```python
#numpyArrayFloat.py
import numpy as np

array = np.array([1, 2., 3, 4, 5])
print(array)
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]

```python
#numpyInstantArrays.py
import numpy as np

range_array = np.arange(0,101)
print(range_array)
```

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.

```python
#numpyInstantArrays.py
import numpy as np

Range_array = np.arange(0,101)
range_array2 = np.arange(0,101, 2)

print(range_array)
print(range_array2)
```

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:

```python
#twoDimensionalListAndNumpyArray.py
import numpy as np

two_dim_list = [[1,2,3,4],[2,3,4,5]]
print(two_dim_list)
```

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 these lists in *two_dim_list* by using a for loop:

```python
#twoDimensionalListAndNumpyArray.py
import numpy as np

twoDimList = [[1,2,3,4],[2,3,4,5]]
print(twoDimList)

for i in range(len(two_dim_list)):
    print(two_dim_list[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:

```python
#twoDimensionalListAndNumpyArray.py
import numpy as np

two_dim_list = [[1,2,3,4],[2,3,4,5]]
print(two_dim_list)

for i in range(len(two_dim_list)):
    print(two_dim_list [i])
    for j in range(len(two_dim_list [i])):
        print(two_dim_list [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:

```
two_dim_array = np.array(two_dim_list)
```

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

```
#twoDimensionalListAndNumpyArray.py
import numpy as np

two_dim_list = [[1,2,3,4],[2,3,4,5]]
print(two_dim_list)
two_dim_array = np.array(two_dim_list)
print(two_dim_array)

for i in range(len(two_dim_list)):
    print(two_dim_list [i])
    print(two_dim_array[i])
    for j in range(len(two_dim_list [i])):
        print(two_dim_list [i][j])
        print(two_dim_array[i][j])
```

Output:

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

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]:

```python
#enterValuesInArray.py
import numpy as np

array = np.zeros((3,3))
print(array)

array[0][2] = 9
array[1][0] = 7
array[2][2] = 3
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:

```python
#zerosOnesAndLike.py
import numpy as np
list_of_lists = [[1,2,3], [4,5,6], [7,8,9]]
array_of_arrays = np.array(list_of_lists)
zeros_like_array = np.zeros_like(list_of_lists)
ones_like_array = np.ones_like(list_of_lists)
empty_like_array = np.empty_like(list_of_lists)

print("list_of_lists:", list_of_lists, sep="\n")
print("array_of_arrays:", array_of_arrays, sep="\n")
print("zeros_like_array:", zeros_like_array, sep="\n")
print("ones_like_array:", ones_like_array, sep="\n")
print("empty_like_array:", empty_like_array, sep="\n")
```

Output:

```
list_of_lists:
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
array_of_arrays:
[[1 2 3]
```

```
 [4 5 6]
 [7 8 9]]
zeros_like_array:
[[0 0 0]
 [0 0 0]
 [0 0 0]]
ones_like_array:
[[1 1 1]
 [1 1 1]
 [1 1 1]]
empty_like_array:
[[         0          0  -           89799145]
 [        34    -1436786320            559]
 [       129     538976288      538976288]]
```

ii. Useful Methods and Values

New Concepts	**Description**
numpy methods (more)	*np.log(), np.log10(),*
numpy values	*np.pi, np.e, np.inf, np.nan*

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.

```python
#numpyLogs.py

vals = np.arange(0, 10001, 100)
ln_vals = np.log(vals)
log10_vals = np.log10(vals)
print(vals)
print(ln_vals)
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:

```
#specialValuesInNumpy.py
```

```python
import numpy as np

pi = np.pi
e = np.e
lne = np.log(e)
infinity = np.inf
null_val = np.nan
print("pi =", pi)
print("e =", e)
print("ln(e)=", lne)
print("infintity:", infinity)
print("null_val:",null_val)
```

Output:

```
pi = 3.141592653589793
e = 2.718281828459045
ln(e)= 1.0
infinity: inf
null_val: 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

New Concepts	**Description**
Boolean indexing	*Numpy* allows a boolean array to be generate in light of a truth condition; e.g., array > 2. The resultant array returns boolean values in that identify whether elements meet the condition or do not. The boolean array generated can be used to identify the subset of the original array that meets the condition.

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:

```python
#evenNumbers.py

even_nums = [i for i in range(100) if i % 2 == 0]
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.

```python
#evenNumbers.py
import numpy as np

even_nums = [i for i in range(100) if i % 2 == 0]
print(even_nums)

even_nums = np.arange(100)
print(even_nums)
index_array = even_nums % 2 == 0
print(index_array)
even_nums = even_nums[index_array]
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  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]
[ True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False  True False  True False  True False  True False
  True False  True False]
[ 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 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

```python
#booleanIndexing.py
```

```python
import random

rand_list = [random.random() * 10 for i in range(7)]
print("rand_list values:", rand_list)
rand_list = [i for i in rand_list if i > 3]
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.

```python
#booleanIndexing.py
import random
import numpy as np

rand_list = [random.random() * 10 for i in range(7)]
print("rand_list values:", rand_list)
rand_list = [i for i in rand_list if i > 3]
print("rand_list values > 3:", rand_list)

rand_array = np.random.randn(7) * 10
print("rand_array:", rand_array)
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

New Concepts	Description

generators (more)	Generators can be used to cycle through keys in a dictionary or even to instantiate a dictionary without crowding the script with several lines of repetative text

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.

```
#dataDict.py
import numpy as np


data_dict = {"0 to 9": np.arange(10),
             "ones": np.ones(10),
             "zeros": np.zeros(10)}
print(data_dict)
```

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.

```
#dataDictpy
import numpy as np


Data_dict = {"1 to 10":np.arange(10),
             "ones":np.ones(10),
             "zeros":np.zeros(10)}
print(data_dict)
print([data_dict[key] for key in data_dict])
```

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*.

```python
#dataDictpy
import numpy as np


data_dict = {"1 to 100":np.arange(10),
             "ones":np.ones(10),
             "zeros":np.zeros(10)}
print(data_dict)
print([data_dict[key] for key in data_dict])
print([key + " " + str(data_dict[key]) for key in data_dict])
```

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]’.

```python
#dataDictpy
import numpy as np


data_dict = {"1 to 100":np.arange(10),
             "ones":np.ones(10),
             "zeros":np.zeros(10)}
print(data_dict)
print([data_dict[key] for key in data_dict])
print([key + " " + str(data_dict[key]) for key in data_dict])
print([key + " " + str(data_dict[key][5:10])  for key in data_dict])
```

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

New Concepts	**Description**

pandas	*Pandas* is the most popular data structure library in python. Its dataframes are particularly popular as they contain many of the same attributes as dictionaries and provide substantially m ore functionality.
pandas data structures	*pd.Series(), pd.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.

```python
#pandasSeriesVsDataFrame.py
import numpy as np
import pandas as pd

dataDict = {"range":np.arange(10)}
dataSeries = pd.Series(dataDict)
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.

```python
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:

```
#pandasSeriesVsDataFrame.py
import numpy as np
import pandas as pd

data_dict = {"range":np.arange(10)}
data_series = pd.Series(data_dict)
print(data_series)
print(data_series["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]
```

Series does not know how to interpret the key and the data associated with each key. Thus, you will want to use a DataFrame if you wish to transform a dictionary with data into a table. It is usually more convenient to work with DataFrame. A DataFrame holds one or multiples 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:

```
#pandasSeriesVsDataFrame.py
import numpy as np
import pandas as pd

data_dict = {"range":np.arange(10)}
data_series = pd.Series(data_dict)
print(data_series)
print(data_series["range"])

data_DF=pd.DataFrame(data_dict)
print(data_DF)
```

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:

```python
#pandasSeriesVsDataFrame.py
import numpy as np
import pandas as pd

data_dict = {"range":np.arange(10)}
data_series = pd.Series(data_dict)
print(data_series)
print(data_series["range"])

data_DF=pd.DataFrame(data_dict)
print(data_DF)
print(data_DF["range"])
```

Output:

```
range    [0, 1, 2, 3, 4, 5, 6, 7, 8, 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.

```
#pandasSeriesVsDataFrame.py
import numpy as np
import pandas as pd

data_dict = {"range":np.arange(10)}
data_series = pd.Series(data_dict)
print(data_series)
print(data_series["range"])

data_DF=pd.DataFrame(data_dict)
print(data_DF)
print(data_DF["range"])
print(data_DF["range"][5:9])
```

Output:

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

```
data_DF.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.

```python
#dictWithIndexAndKey.py
import pandas as pd
import numpy as np
import random
macro_dict = {"GDP":{},
              "Real GDP":{},
              "Price Level":{},
              "Money":{}}

for key in macro_dict:
    for i in range(1990,2018):
        macro_dict[key][i] = np.random.random() * 10000

print(macro_dict)
```

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:

```python
#dictWithIndexAndKey.py
import pandas as pd
import numpy as np
import random
macro_dict = {"GDP":{},
              "Real GDP":{},
              "Price Level":{},
              "Money":{}}

for key in macro_dict:
    for i in range(1990,2018):
        macro_dict[key][i] = np.random.random() * 10000

print(macro_dict)
macro_DF = pd.DataFrame(macro_dict)
print(macro_DF)
```

```
{'GDP': {1990: 7065.235949212676, . . .
              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:

```python
#dictWithIndexAndKey.py
import pandas as pd
import numpy as np
import random
macro_dict = {"GDP":{},
             "Real GDP":{},
             "Price Level":{},
             "Money":{}}

for key in macro_dict:
    for i in range(1990,2018):
        macro_dict[key][i] = np.random.random() * 10000

print(macro_dict)
macro_DF = pd.DataFrame(macro_dict)
macro_DF["Velocity"] =  macro_DF["GDP"] / macro_DF["Money"]
print(macro_DF)
```

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
```

You may choose to call only a slice of a particular column. This may be accomplished in one of two ways. The column name may be called before or after .loc[]. Thus, we can call the same slice for only Real GDP using:

```
macro_DF["Real GDP"].loc[1995:2000]
```

or

```
macro_DF.loc[1995:2000]["Real GDP"]
```

This yields:

```
1995    2701.056522
```

```
1996    2558.332479
1997    8262.518349
1998    1888.230368
1999    7066.159188
2000    3657.004187
Name: Real GDP, dtype: float64
```

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.

c. matplotlib.pyplot

New Concepts	**Description**
matplotlib	Python's most popular data visualization library
matplotlib.pyplot methods	*plt.plot(), plt.scatter(), plt.figure, plt.show(), plt.close(), plt.rcParams(), plt.title(), plt.legend()*
matplotlib wrapper for *pandas*	The *pandas* library can communicate directly with *matplotlib* using a wrapper. This allows matplotlib commands to be called from a *pandas* data frame.
pd.DataFrame() methods	*df.keys()*

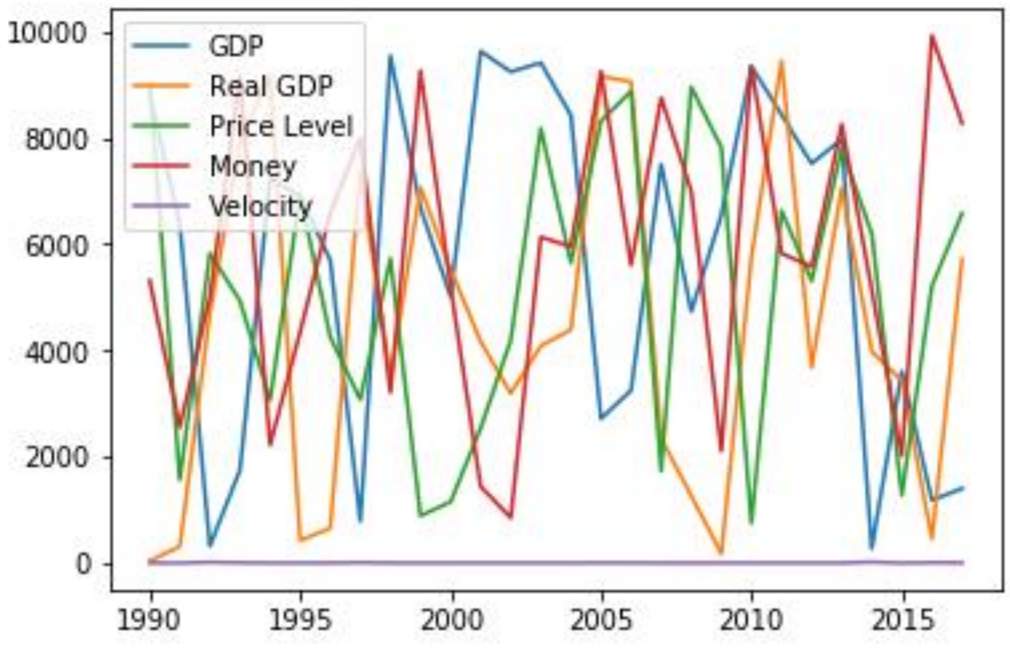
We can use *matplotlib* to plot the data that was just created.working from the same script. There is more than one way to plot data. *Pandas* provides a convenient wrapper that uses *matplotlib.pyplot.plot()* and similar commands with the command:

```
df.plot()
```

Below we build plots using the *matplotlib* wrapper from *pandas*.

```python
#dictWithIndexAndKey.py
import pandas as pd
import numpy as np

macro_dict = {"GDP":{},
              "Real GDP":{},
              "Price Level":{},
              "Money":{}}

for key in macro_dict:
    for i in range(1990,2018):
        macro_dict[key][i] = np.random.random()* 10000

print(macro_dict)
macro_DF = pd.DataFrame(macro_dict)
macro_DF["Velocity"] = macro_DF["Money"] / macro_DF["GDP"]
print(macro_DF)

```

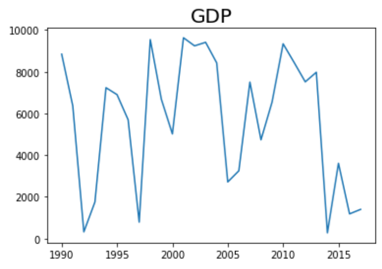
```python
import matplotlib.pyplot as plt
macro_DF.plot.line(legend="best")
```

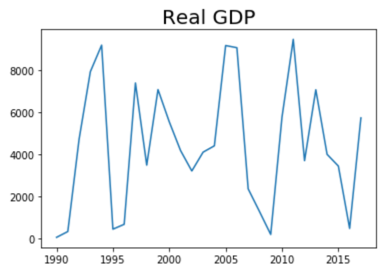


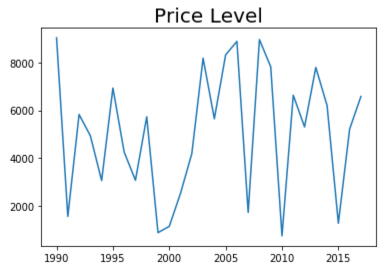
Since the values that we are using are most random, the graph is a bit cluttered. This means that there is no good place within the graph to place the legend. It may help to create graphs for each key one at a time. For this, we will use a *for loop* that calls each *key* (i.e., column name) one at a time. For each key called, we can plot the data associated with the key in a single graph.

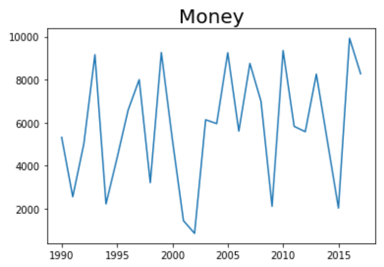
```python
#dictWithIndexAndKey.py
. . .

for key in macro_DF:
    macro_DF[key].plot.line()
    plt.title(key,fontsize=20)
    plt.show()
    plt.close()
```

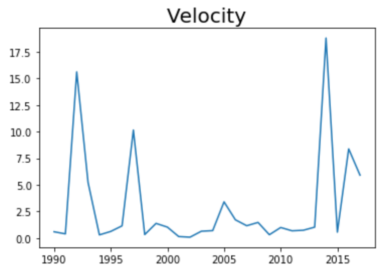
The following graphs are generated in the console:







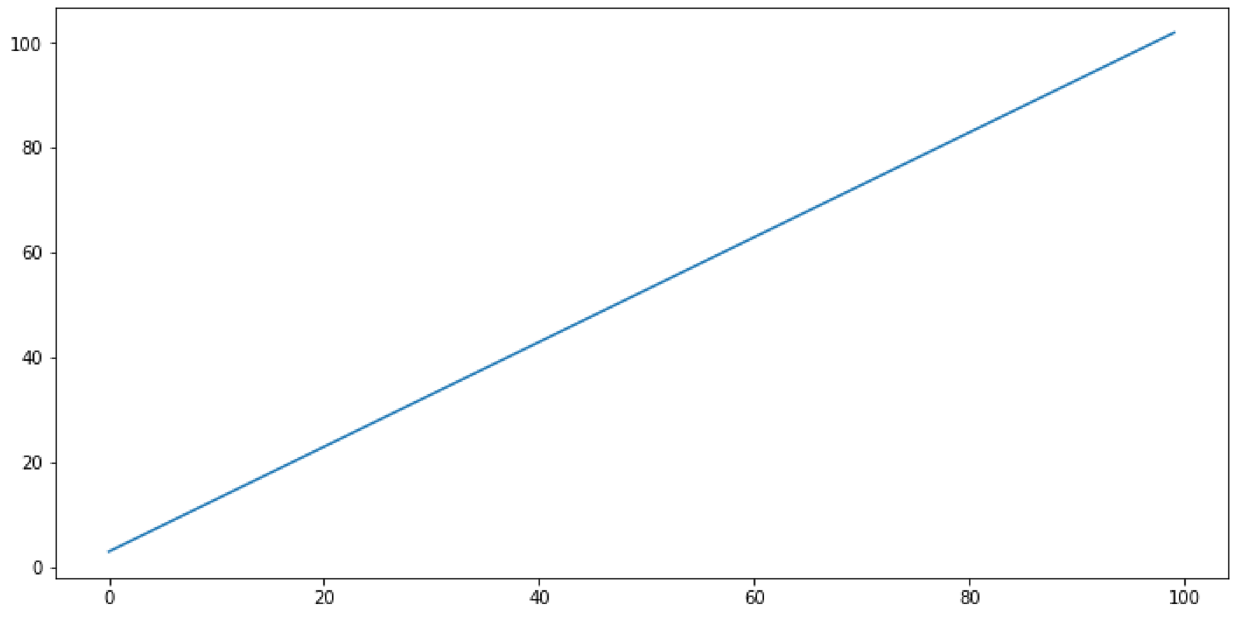




`

The graphs from the last example were created by using the matplotlib options that are integrated into pandas. While this a convenient approach, it may not always be the most appropriate means for creating the ideal visual. In the next example we will continue to explore matplotlib, generating a line and platting random points around that line. First, let’s plot the line, calling plt.figure() beforehand so that we can determine the size of the figure generated. We choose the dimensions (12,6) below.

```
#lineAndRandomPoints.py
import numpy as np
import matplotlib.pyplot as plt

line = np.array([i + 3 for i in range(100)])
figure = plt.figure(figsize = (12,6))
plt.plot(line)

plt.show()
```

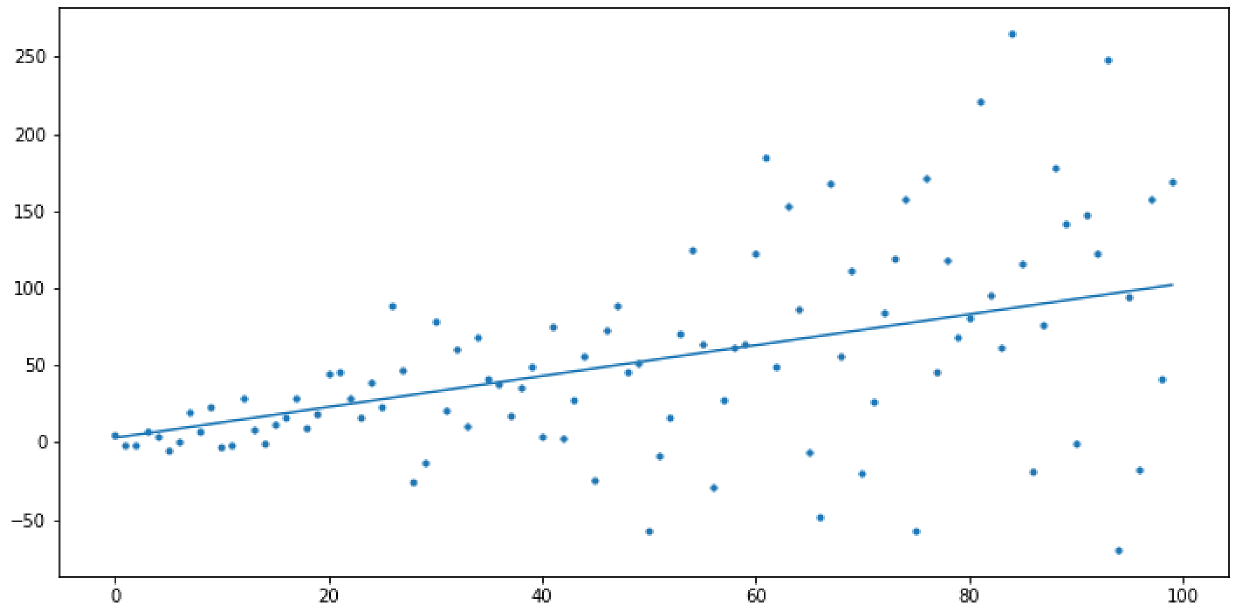


The line created is the true line. We will generate points from normal distributions whose median values are indicated by the line using *normalvariate()*:

```
points.append(random.normalvariate(mu, sigma))
```

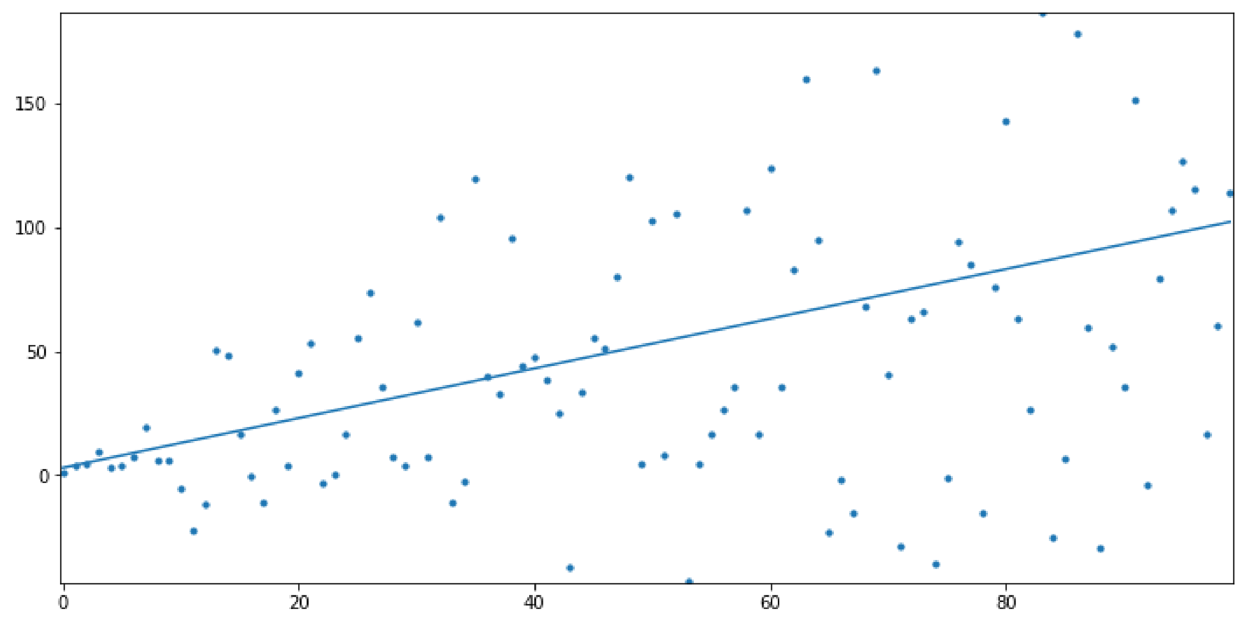
Here, mu defines the expectation –the mean and, by implication, the median – of the distribution and. *Sigma* indicates the standard deviation. We will pass values from the line to both, thus increasing the mean and variance of the distribution.

```python
#lineAndRandomPoints.py
import numpy as np
import matplotlib.pyplot as plt
import random

line = np.array([i + 3 for i in range(100)])
points = []
for point in line:
    points.append(random.normalvariate(point,point))

figure = plt.figure(figsize = (12,6))
plt.plot(line)
plt.scatter(np.arange(len(points)),points, s = 10)

plt.show()
```



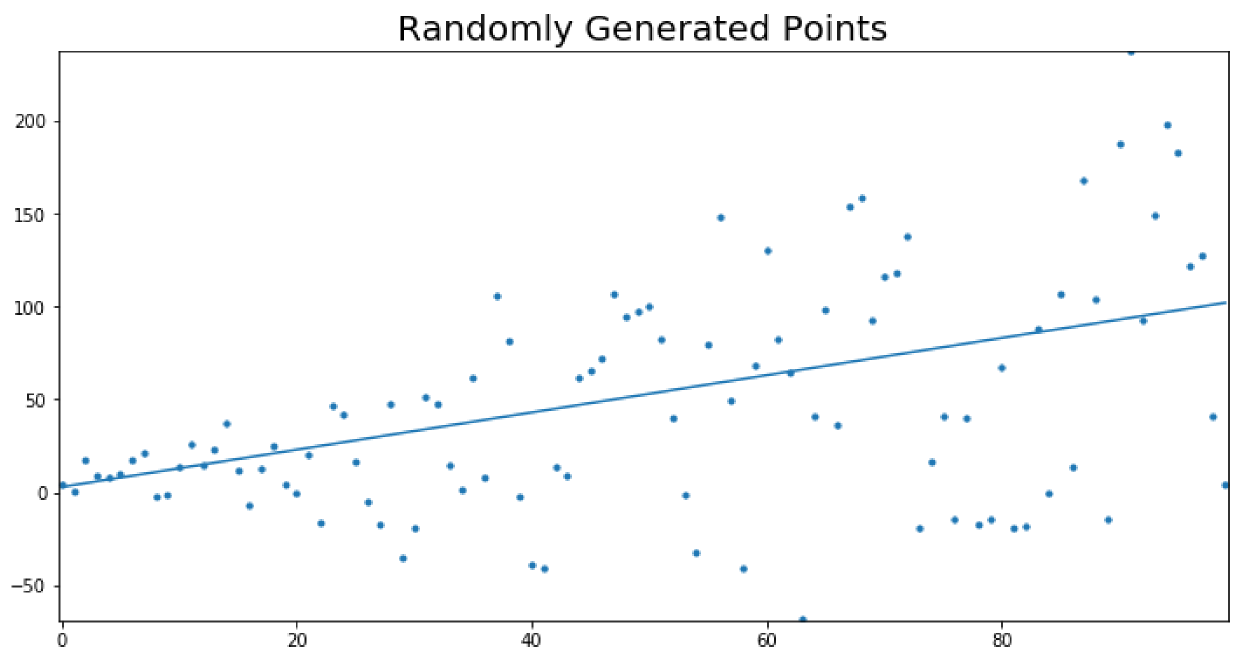
Matplotlib automatically leaves a margin behind. Reduce the margins using *plt.rcParams*. This allows access to variables that define this margin. Reduce the margin to *0*, as reflected in line 14 and 15.

```
1.  #lineAndRandomPoints.py
2.  import numpy as np
3.  import matplotlib.pyplot as plt
4.  import random
5.  
6.  line = np.array([i + 3 for i in range(100)])
7.  points = []
8.  for point in line:
9.      points.append(random.normalvariate(0,point) + point)
10. figure = plt.figure(figsize = (12,6))
11. plt.rcParams['axes.xmargin'] = 0
12. plt.rcParams['axes.ymargin'] = 0
13. plt.plot(line)
14. plt.scatter(np.arange(len(points)), points, s = 10)
15. plt.show()
```



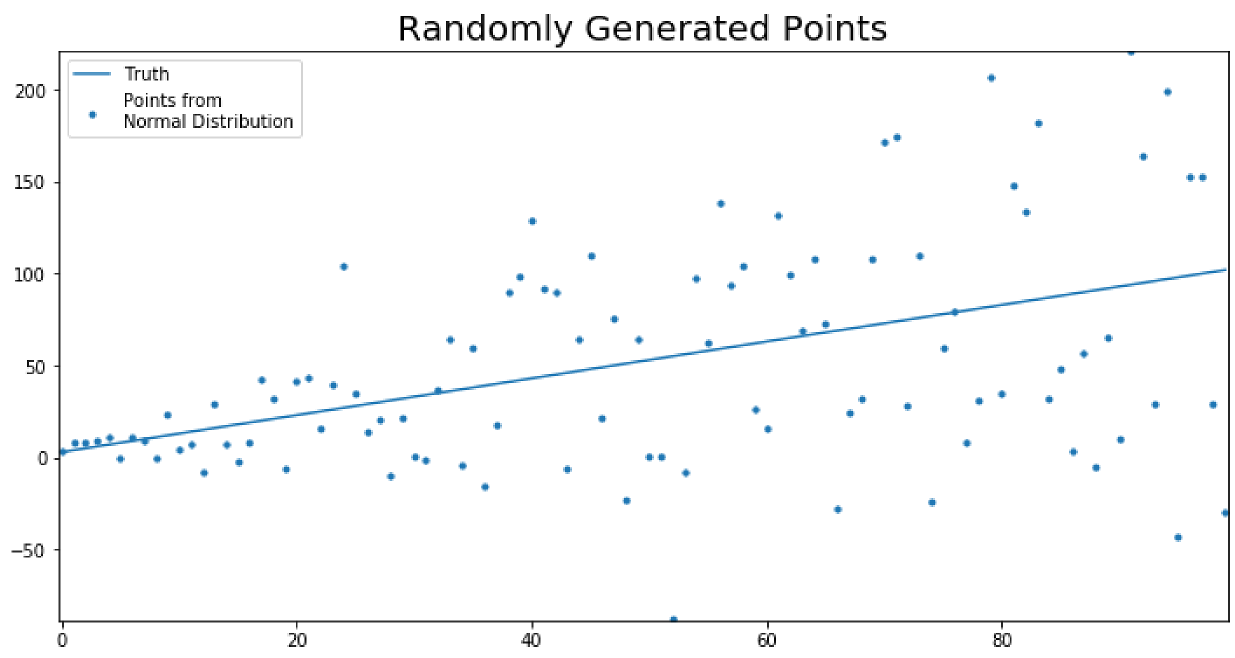
A graph should have a title, labels, or both. Add a title using *plt.title()*. The size of this text is defined by *fontsize*.

```python
#lineAndRandomPoints.py
import numpy as np
import matplotlib.pyplot as plt
import random

line = np.array([i + 3 for i in range(100)])
points = []
for point in line:
    points.append(random.normalvariate(0,point) + point)
figure = plt.figure(figsize = (12,6))
plt.rcParams['axes.xmargin'] = 0
plt.rcParams['axes.ymargin'] = 0
plt.plot(line, label = "Truth")
plt.scatter(np.arange(len(points)), points, s = 10,
            label = "Points from\nNormal Distribution")
plt.title("Randomly Generated Points", fontsize = 16)

plt.show()
```



Finally, we add labels to each of the types of objects in the graph. This is accomplished by adding *label* = *label_name* to *plt.plot()* and *plt.scatter()*.

```python
#lineAndRandomPoints.py
import numpy as np
import matplotlib.pyplot as plt
import random

line = np.array([i + 3 for i in range(100)])
points = []
for point in line:
    points.append(random.normalvariate(0,point) + point)
figure = plt.figure(figsize = (12,6))
plt.rcParams['axes.xmargin'] = 0
plt.rcParams['axes.ymargin'] = 0
plt.plot(line, label = "Truth")
plt.scatter(np.arange(len(points)), points, s = 10,
            label = "Points from\nNormal Distribution")
plt.title("Randomly Generated Points", fontsize = 16)
plt.legend(loc="Best")
plt.show()
```

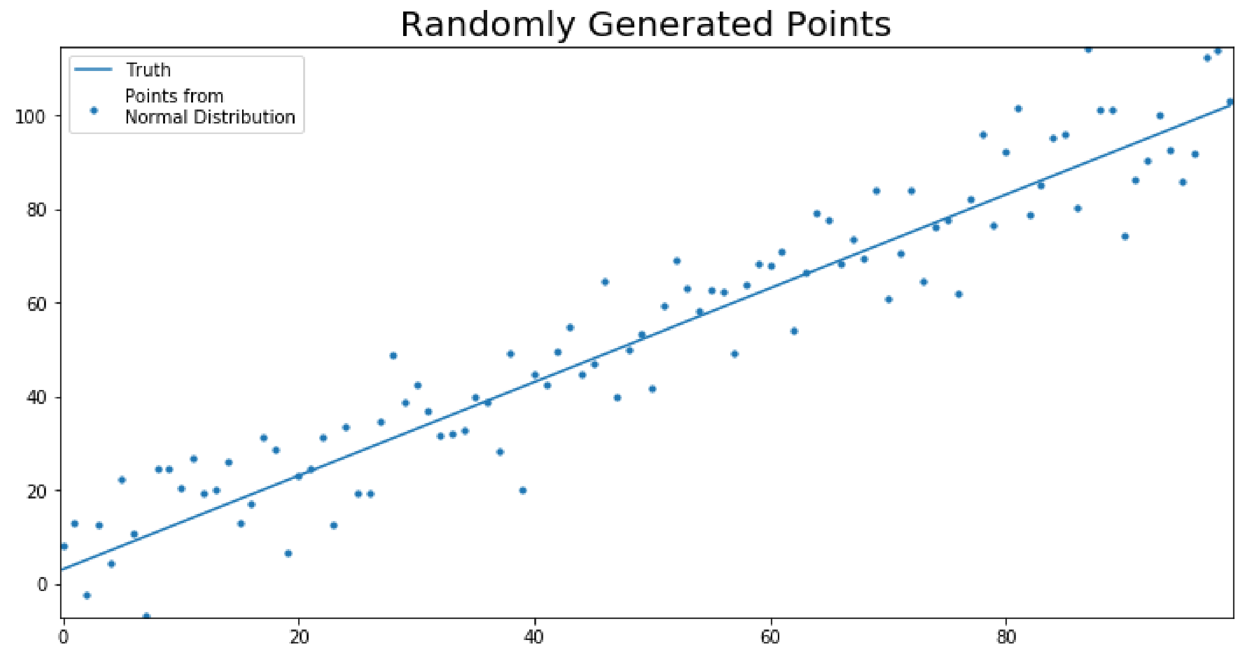


It may be convenient to create a functions that allow for control of a few features of the line, points, and visualizations. For simplicity, the standard deviation governing the creation of the scatter plot will now be a constant.

```python
#lineAndRandomPointsFunction.py
import numpy as np
import matplotlib.pyplot as plt
import random

def build_random_data_with_line(y_int, slope, SD = 1):

    # line is defined by the y_intercept and slope
    line = np.array([slope * (i + y_int) for i in range(100)])
    points = []
    for point in line:
        points.append(random.normalvariate(point, SD))

    return line, points

def plot_line(line, points, line_name = "Truth",
              title = "Randomly Generated Points"):

    figure = plt.figure(figsize = (12,6))
    plt.rcParams['axes.ymargin'] = 0
    plt.rcParams['axes.xmargin'] = 0
    plt.plot(line, label = line_name)
    plt.scatter(np.arange(len(points)),points, s = 10,
                label = "Points from\nNormal Distribution")
    plt.title(title, fontsize = 20)
    plt.legend(loc="best")
    plt.show()

```

```
line, points = build_random_data_with_line(y_int = 3, slope = 1, SD = 10)
plot_line(line, points)
```

Output:



What other feature would be useful to include in these functions? What about parameters for different font sizes, size and colors of points and lines in scatter plots?

Exercises:

1. Create an array with (100) elements using np.arange()

2. Create two numpy arrays whose values are neither all 0 or 1. Multiply the two arrays and save the result i.e., result = array1 * array2.

3. Create a dictionary where each key points to a unique numpy array. All arrays should be of the same length. Then transform that dictionary into a pandas dataframe.

4. Import a data set for your project in pandas. You may use whatever means you prefer to import the data. Print the dataframe.

5. Create a new column of data derived from the data that you have already imported. hint: this may be a ratio or whatever transformation that you think would be useful...

6. Use a for loop to print individually each column in the dataframe.

7. Create a function that plots 2 to 3 different variables from the dataframe. The function may look something like "def plot_data(df,names, etc...). Each plot should include a legend and title.

8. Create a bar graph plot using the data you have imported.

9. On any of the graphs you have created, change the font size, axis labels, tick labels, color, linewidth, and so forth. Adjust these parameters to make the visualization aesthetically appealing..

6. Importing, Cleaning, and Analyzing Data

a. Importing, managing, and analyzing data

New Concepts	Description
pandas methods (more)	*pd.read_csv(), pd.to_numeric()*
pd.DataFrame() methods	*df.values, df.dropna(), df[key].dtype, df.to_csv(), df.ix, df.sort_index()*

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.*

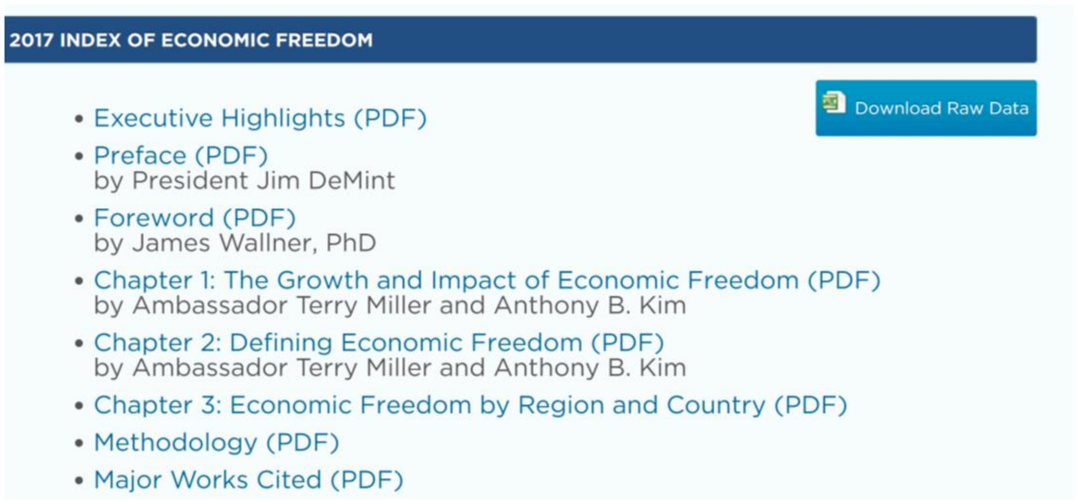

2017 INDEX OF ECONOMIC FREEDOM
Download Raw Data
Executive Highlights (PDF)
Preface (PDF)
by President Jim DeMint
Foreword (PDF)
by James Wallner, PhD
Chapter 1: The Growth and Impact of Economic Freedom (PDF)
by Ambassador Terry Miller and Anthony B. Kim
Chapter 2: Defining Economic Freedom (PDF)
by Ambassador Terry Miller and Anthony B. Kim
Chapter 3: Economic Freedom by Region and Country (PDF)
Methodology (PDF)
Major Works Cited (PDF)

Figure 6.1

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.

a. 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 several other libraries, as well as stats – the *stats.py* file we saved in the same folder – as we will use these later.

```
#economicFreedomStats.py
import pandas as pd
import stats

data = pd.read_excel("index2017_data.xls", index_col = [1])
```

If you use shape, you can check the number of rows and columns:

```
data.shape
```

Output:

```
(186, 33)
```

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…

We will start by removing columns with data that will not be useful for this exercise. And we will only included row (i.e., countries) with data for every column:

```
#economicFreedomStats.py
import pandas as pd
import stats

data = pd.read_excel("index2017_data.xls", index_col = [1])

# some columns are not needed for the purposes of this exercise, so
# we drop them
skip_keys = ["CountryID", "Region", "WEBNAME", "Country"]
data_for_stats = data.drop(skip_keys, axis = 1)
#Drop rows that do not include observations for every category
data_for_stats = data_for_stats.dropna(thresh = len(data_for_stats.columns))
```

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. We will use an instance of *Stats()* to fill these dictionaries with summary statistics.

```
#economicFreedomStats.py
import pandas as pd
import stats
```

```
4.
5.  data = pd.read_excel("index2017_data.xls", index_col = [1])
6.
7.  # some columns are not needed for the purposes of this exercise, so
8.  # we drop them
9.  skip_keys = ["CountryID", "Region", "WEBNAME", "Country"]
10. data_for_stats = data.drop(skip_keys, axis = 1)
11. #Drop rows that do not include observations for every category
12. data_for_stats = data_for_stats.dropna(thresh = len(data_for_stats.columns))
13.
14. # Next, we create dictionaries that will hold statistics for each variable
15. # or pair of variables in the case of cov and corr
16. stat = stats.Stats()
17. stats_dict = {}
18. cov_dict = {}
19. corr_dict = {}
```

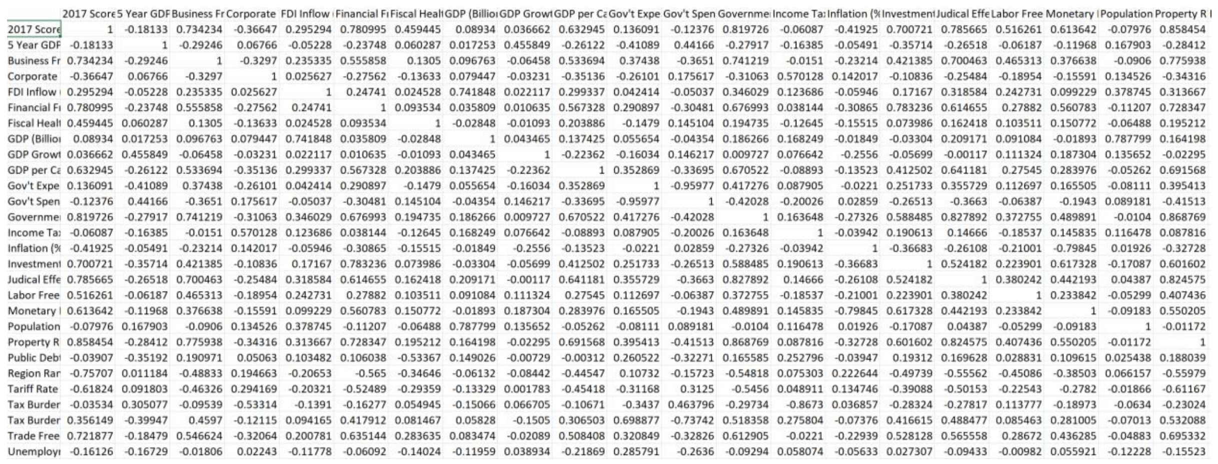
We are ready to fill the dictionaries with summary statistics. We will use two for-loops. The first for-loop will be used to fill the *statsDict* with summary statistics requiring only one variable. The second for-loop, which is within the first for-loop, is used to create the *corrDict* and *cov_dict*. 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 you enter *cov_dict[“2017 Score”][“Property Rights”]* after *cov_dict* has been constructed, this will call the covariance of these two variables. The two for-loops cycle through listof variables, thus calling every combination of variables (twice) to create a matrix of covariance and correlation values.

```
1.  #economicFreedomStats.py
2.  . . .
19. corr_dict = {}
20.
21. for key1 in data_for_stats:
22.     # to use the functions from stats requires that we call lists, not Series
23.     # so a list of values is create for each variable (key1) in the dataframe
24.     vec1 = data_for_stats[key1]
25.     stats_dict[key1] = {}
26.     stats_dict[key1]["mean"] = stat.mean(vec1)
27.     stats_dict[key1]["median"] = stat.median(vec1)
28.     stats_dict[key1]["variance"] = stat.variance(vec1)
29.     stats_dict[key1]["standard deviation"] = stat.SD(vec1, sample = True)
30.     stats_dict[key1]["skewness"] = stat.skewness(vec1, sample = True)
31.     stats_dict[key1]["kurtosis"] = stat.kurtosis(vec1, sample = True)
32.     cov_dict[key1] = {}
33.     corr_dict[key1] = {}
34.     for key2 in data_for_stats:
35.         vec2 = data_for_stats[key2]
36.         cov_dict[key1][key2] = stat.covariance(vec1, vec2, sample = True)
37.         corr_dict[key1][key2] = stat.correlation(vec1, vec2)
```

Finally, if we want to see these tables in a convenient format (i.e., Figure 6.2), 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.  . . .
```

```python
#convert stats, cov, and corr dictionaries to pandas DataFrames
stats_DF = pd.DataFrame(stats_dict)
cov_DF = pd.DataFrame(cov_dict).sort_index(axis=1)
corr_DF = pd.DataFrame(corr_dict).sort_index(axis=1)

# output DataFrames to CSV
stats_DF.to_csv("econFreedomStatsByCategory.csv")
cov_DF.to_csv("econFreedomCovMatrix.csv")
corr_DF.to_csv("econFreedomCorrMatrix.csv")
data_for_stats.to_csv("cleanedEconFreedomData.csv")
```



	2017 Score	5 Year GDF	Business Fr	Corporate	FDI Inflow	Financial Fi	Fiscal Healt	GDP (Billio	GDP Growt	GDP per Ca	Gov't Expe	Gov't Spen	Governme	Income Ta	Inflation (%	Investment	Judical Effe	Labor Free	Monetary	Population	Property R
2017 Score	1	-0.18133	0.734234	-0.36647	0.295294	0.780995	0.459445	0.08934	0.036662	0.632945	0.136091	-0.12376	0.819726	-0.06087	-0.41925	0.700721	0.785665	0.516261	0.613642	-0.07976	0.858454
5 Year GDF	-0.18133	1	-0.29246	0.06766	-0.05228	-0.23748	0.060287	0.017253	0.455849	-0.26122	-0.41089	0.44166	-0.27917	-0.16385	-0.05491	-0.35714	-0.26518	-0.06187	-0.11968	0.167903	-0.28412
Business Fr	0.734234	-0.29246	1	-0.3297	0.235335	0.555858	0.1305	0.096763	-0.06458	0.533694	0.37438	-0.3651	0.741219	-0.0151	-0.23214	0.421385	0.700463	0.465313	0.376638	-0.0906	0.775938
Corporate	-0.36647	0.06766	-0.3297	1	0.025627	-0.27562	-0.13633	0.079447	-0.03231	-0.35136	-0.26101	0.175617	-0.31063	0.570128	0.142017	-0.10836	-0.25484	-0.18954	-0.15591	0.134526	-0.34316
FDI Inflow	0.295294	-0.05228	0.235335	0.025627	1	0.24741	0.024528	0.741848	0.022117	0.299337	0.042414	-0.05037	0.346029	0.123686	-0.05946	0.17167	0.318584	0.242731	0.099229	0.378745	0.313667
Financial Fi	0.780995	-0.23748	0.555858	-0.27562	0.24741	1	0.093534	0.035809	0.010635	0.567328	0.290897	-0.30481	0.676993	0.038144	-0.30865	0.783236	0.614655	0.27882	0.560783	-0.11207	0.728347
Fiscal Healt	0.459445	0.060287	0.1305	-0.13633	0.024528	0.093534	1	-0.02848	-0.01093	0.203886	-0.1479	0.145104	0.194735	-0.12645	-0.15515	0.073986	0.162418	0.103511	0.150772	-0.06488	0.195212
GDP (Billio	0.08934	0.017253	0.096763	0.079447	0.741848	0.035809	-0.02848	1	0.043465	0.137425	0.055654	-0.04354	0.186266	0.168249	-0.01849	-0.03304	0.209171	0.091084	-0.01893	0.787799	0.164198
GDP Growt	0.036662	0.455849	-0.06458	-0.03231	0.022117	0.010635	-0.01093	0.043465	1	-0.22362	-0.16034	0.146217	0.009727	0.076642	-0.2556	-0.05699	-0.00117	0.111324	0.187304	0.135652	-0.02295
GDP per Ca	0.632945	-0.26122	0.533694	-0.35136	0.299337	0.567328	0.203886	0.137425	-0.22362	1	0.352869	-0.33695	0.670522	-0.08893	-0.13523	0.412502	0.641181	0.27545	0.283976	-0.05262	0.691568
Gov't Expe	0.136091	-0.41089	0.37438	-0.26101	0.042414	0.290897	-0.1479	0.055654	-0.16034	0.352869	1	-0.95977	0.417276	0.087905	-0.0221	0.251733	0.355729	0.112697	0.165505	-0.08111	0.395413
Gov't Spen	-0.12376	0.44166	-0.3651	0.175617	-0.05037	-0.30481	0.145104	-0.04354	0.146217	-0.33695	-0.95977	1	-0.42028	-0.20026	0.02859	-0.26513	-0.3663	-0.06387	-0.1943	0.089181	-0.41513
Governme	0.819726	-0.27917	0.741219	-0.31063	0.346029	0.676993	0.194735	0.186266	0.009727	0.670522	0.417276	-0.42028	1	0.163648	-0.27326	0.588485	0.827892	0.372755	0.489891	-0.0104	0.868769
Income Ta	-0.06087	-0.16385	-0.0151	0.570128	0.123686	0.038144	-0.12645	0.168249	0.076642	-0.08893	0.087905	-0.20026	0.163648	1	-0.03942	0.190613	0.14666	-0.18537	0.145835	0.116478	0.087816
Inflation (%	-0.41925	-0.05491	-0.23214	0.142017	-0.05946	-0.30865	-0.15515	-0.01849	-0.2556	-0.13523	-0.0221	0.02859	-0.27326	-0.03942	1	-0.36683	-0.26108	-0.21001	-0.79845	0.01926	-0.32728
Investment	0.700721	-0.35714	0.421385	-0.10836	0.17167	0.783236	0.073986	-0.03304	-0.05699	0.412502	0.251733	-0.26513	0.588485	0.190613	-0.36683	1	0.524182	0.223901	0.617328	-0.17087	0.601602
Judical Effe	0.785665	-0.26518	0.700463	-0.25484	0.318584	0.614655	0.162418	0.209171	-0.00117	0.641181	0.355729	-0.3663	0.827892	0.14666	-0.26108	0.524182	1	0.380242	0.442193	0.04387	0.824575
Labor Free	0.516261	-0.06187	0.465313	-0.18954	0.242731	0.27882	0.103511	0.091084	0.111324	0.27545	0.112697	-0.06387	0.372755	-0.18537	-0.21001	0.223901	0.380242	1	0.233842	-0.05299	0.407436
Monetary	0.613642	-0.11968	0.376638	-0.15591	0.099229	0.560783	0.150772	-0.01893	0.187304	0.283976	0.165505	-0.1943	0.489891	0.145835	-0.79845	0.617328	0.442193	0.233842	1	-0.09183	0.550205
Population	-0.07976	0.167903	-0.0906	0.134526	0.378745	-0.11207	-0.06488	0.787799	0.135652	-0.05262	-0.08111	0.089181	-0.0104	0.116478	0.01926	-0.17087	0.04387	-0.05299	-0.09183	1	-0.01172
Property R	0.858454	-0.28412	0.775938	-0.34316	0.313667	0.728347	0.195212	0.164198	-0.02295	0.691568	0.395413	-0.41513	0.868769	0.087816	-0.32728	0.601602	0.824575	0.407436	0.550205	-0.01172	1
Public Debt	-0.03907	-0.35192	0.190971	0.05063	0.103482	0.106038	-0.53367	0.149026	-0.00729	-0.00312	0.260522	-0.32271	0.165585	0.252796	-0.03947	0.19312	0.169628	0.028831	0.109615	0.025438	0.188039
Region Rar	-0.75707	0.011184	-0.48833	0.194663	-0.20653	-0.565	-0.34646	-0.06132	-0.08442	-0.44547	0.10732	-0.15723	-0.54818	0.075303	0.222644	-0.49739	-0.55562	-0.45086	-0.38503	0.066157	-0.55979
Tariff Rate	-0.61824	0.091803	-0.46326	0.294169	-0.20321	-0.52489	-0.29359	-0.13329	0.001783	-0.45418	-0.31168	0.3125	-0.5456	0.048911	0.134746	-0.39088	-0.50153	-0.22543	-0.2782	-0.01866	-0.61167
Tax Burder	-0.03534	0.305077	-0.09539	-0.53314	-0.1391	-0.16277	0.054945	-0.15066	0.066705	-0.10671	-0.3437	0.463796	-0.29734	-0.8673	0.036857	-0.28324	-0.27817	0.113777	-0.18973	-0.0634	-0.23024
Tax Burder	0.356149	-0.39947	0.4597	-0.12115	0.094165	0.417912	0.081467	0.05828	-0.1505	0.306503	0.698877	-0.73742	0.518358	0.275804	-0.07376	0.416615	0.488477	0.085463	0.281005	-0.07013	0.532088
Trade Free	0.721877	-0.18479	0.546624	-0.32064	0.200781	0.635144	0.283635	0.083474	-0.02089	0.508408	0.320849	-0.32826	0.612905	-0.0221	-0.22939	0.528128	0.565558	0.28672	0.436285	-0.04883	0.695332
Unemployi	-0.16126	-0.16729	-0.01806	0.02243	-0.11778	-0.06092	-0.14024	-0.11959	0.038934	-0.21869	0.285791	-0.2636	-0.09294	0.058074	-0.05633	0.027307	-0.09433	-0.00982	0.055921	-0.12228	-0.15523

Correlation Matrix in CSV Format

Figure 2.2

b. Visualizing Data

New Concepts	Description
matplotlib (more)	*PdfPages(), plt.subplots(), plt.xticks(), plt.setp(), fig.colorbar(), ax.imshow(), plt.tight_layout()*
Pandas methods	*pd.plotting.scatter_matrix(), df.ix[index]*

Next we will visualize the data. We will create scatterplots that use color to indicate a 3rd dimension. Then we wil create a heatmap that visualizes a subset of the correlation matrix. Finally, we will create a single figure that shows the scatter plots of each pair of this subset and the distribution of each variable along the diagonal of the figure.

Most of the images created in this exercise will be saved in a single PDF, with one image on each page. We do this by calling PdfPages, which is imported from *matplotlib.backends.backend_pdf*. Each image will be saved as an object, *fig*, and that object will be passed to the savefig method from this library.

In preparation for creating these images, the default *fontsize* is set to 24 and a subset of variables is chosen to be represented. If you prefer, you may add to or change this list. Keep in mind that changes in the dimension of the latter two types of visulizations may affect the formatting.

```python
#econFreedomVisualizations.py
```

```python
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

data = pd.read_csv("cleanedEconFreedomData.csv", index_col = ["Country Name"])
# Save plots in a pdf using PdfPages
pp = PdfPages("Economic Freedom Plots.pdf")
# Set size of font used unless otherwise specified
plt.rcParams.update({"font.size": 26})
# select subset of variables to visualize in scatter plot
scatter_cats = ["World Rank", "2017 Score", "Property Rights",
                "Judical Effectiveness", "5 Year GDP Growth Rate (%)",
                "GDP per Capita (PPP)"]
select_data = data[scatter_cats]
```

i. Scatter Plots

By cycling through the same list of keys 3 times, we are able to create scatter plots that represent different sets of variables. The if statements prevent the creation of plots that use the same variable on more than one axis. (How would you prevent the same pair of variables appearing on the x and y and vice versa?) Plots are created for every set of variables in the dataframe.

When creating a visualization, the script may identify the attributes of text, the axes, the dimensions of the visualization, and so forth. The primary method, *.plot.scatter()*, also includes a parameter, alpha, that selects the level of transparancy (from most transparent, 0, to least transparent, 1). The colormap setting chosen is "viridis". Other settings can be viewed at: https://matplotlib.org/users/colormaps.html.

Finally, the program identifies the figure to be used by passing *ax = ax* to *DataFrame.plot.scatter()*. Each image is saved as a figure using the *.get_figure()* method that is included in *pd.DataFrame.plot()*. This is then passed to the *PdfPages* variable that was saved in line 111.

```python
#econFreedomVisualizations.py
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

def color_dim_scatter(data, pp):
# function uses color as dimension in scatter plot
    for key1 in data:
        for key2 in data:
            # do not use same variable for x and y dimension
            if key1 != key2:
                for key3 in data:
                    # do not create visualization if key1 or key2
                    # equals key3
                    if key1 != key3 and key2 != key3:
                        # Choose figure size and save ax as object
                        fig, ax = plt.subplots(figsize = (20, 20))
                        # each point represents an observation with 3 different
                        # values: key1 on the horiz ax, key2 on the vert ax,
                        # and key3 as color
                        data.plot.scatter(x = key1, y = key2, c = key3, s = 50,
                                          alpha = .7, colormap = "viridis",
                                          ax = ax)
                        # Make values on x-axis vertical
```

```
25.                         plt.xticks(rotation = 90)
26.                         # Remove tick lines
27.                         plt.setp(ax.get_xticklines(), visible = False)
28.                         plt.setp(ax.get_yticklines(), visible = False)
29.                         plt.show()
30.                         pp.savefig(fig, bbox_inches = "tight")
31.                         plt.close()
32.
33.
34. data = pd.read_csv("cleanedEconFreedomData.csv", index_col = ["Country Name"])
35. # Save plots in a pdf using PdfPages
36. pp = PdfPages("Economic Freedom Plots.pdf")
37. # Set size of font used unless otherwise specified
38. plt.rcParams.update({"font.size": 26})
39. # select subste of variables to visualize in scatter plot
40. scatter_cats = ["World Rank", "2017 Score", "Property Rights",
41.                 "Judical Effectiveness", "5 Year GDP Growth Rate (%)",
42.                 "GDP per Capita (PPP)"]
43. select_data = data[scatter_cats]
44. color_dim_scatter(select_data, pp)
45. pp.close()
```

Conole:

```
pp.close()
```

Figures of the following format are created and saved in the pdf that was created in line 80.

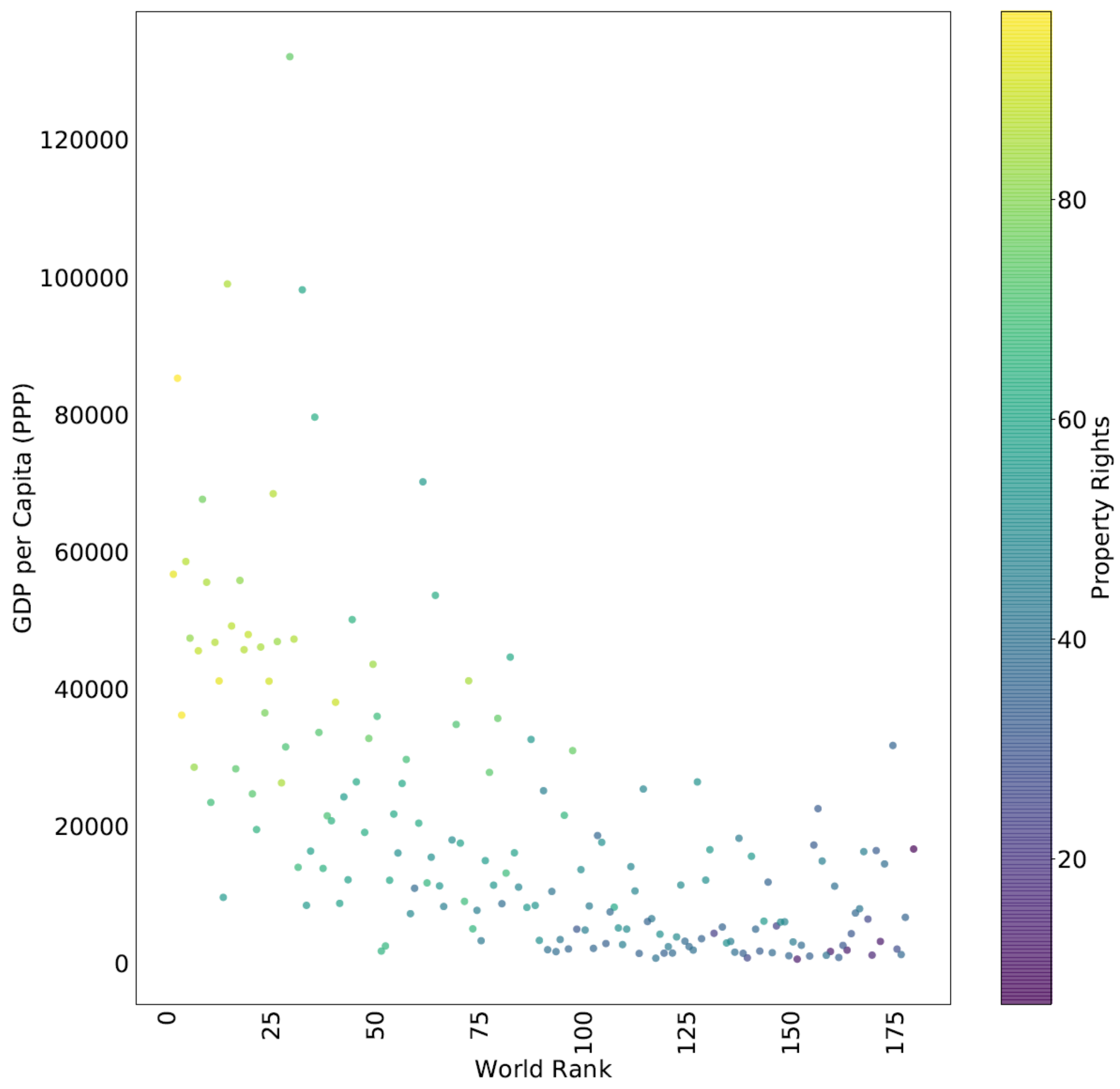


Figure 2.2

ii. Correlation Heatmap

Next, we will create a heatmap that shows represents the correlation matrix. This is a relatively straightforward task since we have already organized the data is it will appear in this heatmap. We create another figure using *plt.subplots()*. The ax variable is used to call imshow, *ax.imshow(df)*, which is creates the heatmap. The labels, however, are not automatically generated. We create a variable that holds the *labels* and pass, along with an empty list of the same length to *plt.xticks()* and *plt.yticks()*. Passing the title “Correlation” to *plt.title()* generates a title at the top of the plot.

```
#econFreedomVisualizations.py
```

```python
import numpy as np
. . .

def corr_matrix_heatmap(data, pp):
    #Create a figure to visualize a corr matrix
    fig, ax = plt.subplots(figsize=(20,20))
    # use ax.imshow() to create a heatmap of correlation values
    # seismic mapping shows negative values as blue and positive values as red
    im = ax.imshow(data, cmap = "seismic")
    # create a list of labels, stacking each word in a label by replacing " "
    # with "\n"
    labels = data.keys()
    num_vars = len(labels)
    tick_labels = [lab.replace(" ", "\n") for lab in labels]
    # adjust font size according to the number of variables visualized
    tick_font_size = 120 / num_vars
    val_font_size = 200 / num_vars
    # prepare space for label of each column
    x_ticks = np.arange(num_vars)
    # select labels and rotate them 90 degrees so that they are vertical
    plt.xticks(x_ticks, tick_labels, fontsize = tick_font_size, rotation = 90)
    # prepare space for label of each row
    y_ticks = np.arange(len(labels))
    # select labels
    plt.yticks(y_ticks, tick_labels, fontsize = tick_font_size)
    # show values in each tile of the heatmap
    for i in range(len(labels)):
        for j in range(len(labels)):
            text = ax.text(i, j, str(round(data.values[i][j],2)),
                           fontsize= val_font_size, ha="center",
                           va="center", color = "w")
    #Create title with Times New Roman Font
    title_font = {"fontname":"Times New Roman"}
    plt.title("Correlation", fontsize = 50, **title_font)
    #Call scale to show value of colors
    cbar = fig.colorbar(im)
    cbar.set_clim(-1,1)
    plt.show()
    pp.savefig(fig, bbox_inches="tight")
    plt.close()


data = pd.read_csv("cleanedEconFreedomData.csv", index_col = ["Country Name"])
corr_data = pd.read_csv("econFreedomCorrMatrix.csv", index_col = [0])
# Save plots in a pdf using PdfPages
pp = PdfPages("Economic Freedom Plots.pdf")
# Set size of font used unless otherwise specified
plt.rcParams.update({"font.size": 26})
# select subste of variables to visualize in scatter plot
scatter_cats = ["World Rank", "2017 Score", "Property Rights",
                "Judical Effectiveness", "5 Year GDP Growth Rate (%)",
                "GDP per Capita (PPP)"]
select_data = data[scatter_cats]
select_corr_data = corr_data.loc[scatter_cats][scatter_cats]
color_dim_scatter(select_data, pp)
corr_matrix_heatmap(select_corr_data, pp)
pp.close()
```

The resulting heatmap provides a powerful presentation of the data. Both the actual value of correlation and the color are shown in the same tile, thereby reinforcing the information conveyed to the viewer. It is worthwshile to spend time with a visualization, imagining what changes or additions can be made in order to improve clarity for the viewer. Aesthetics might not be everything, but when time a constraint, a well crafted visualizatoin can convey a significant amount of knowledge.

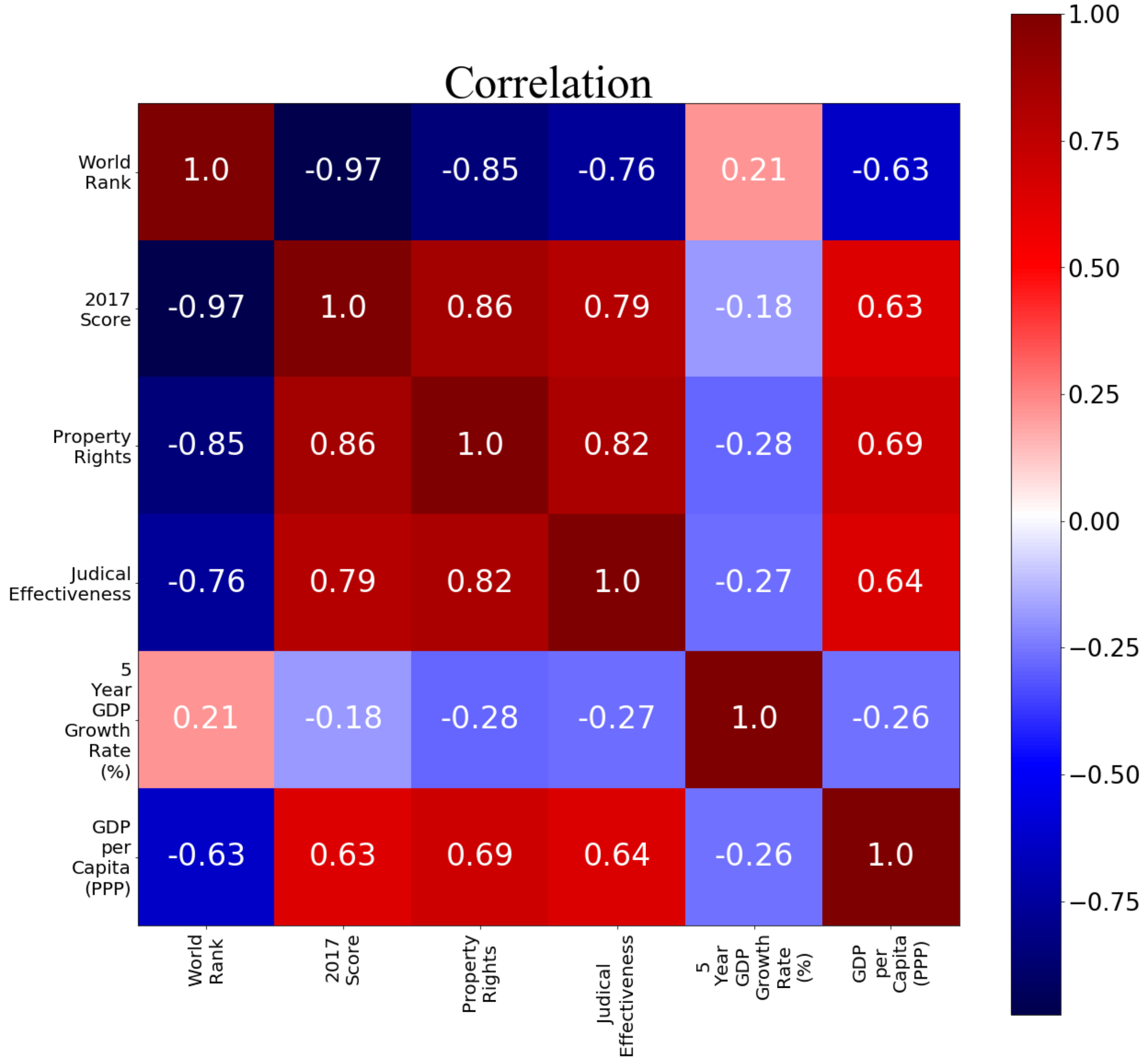
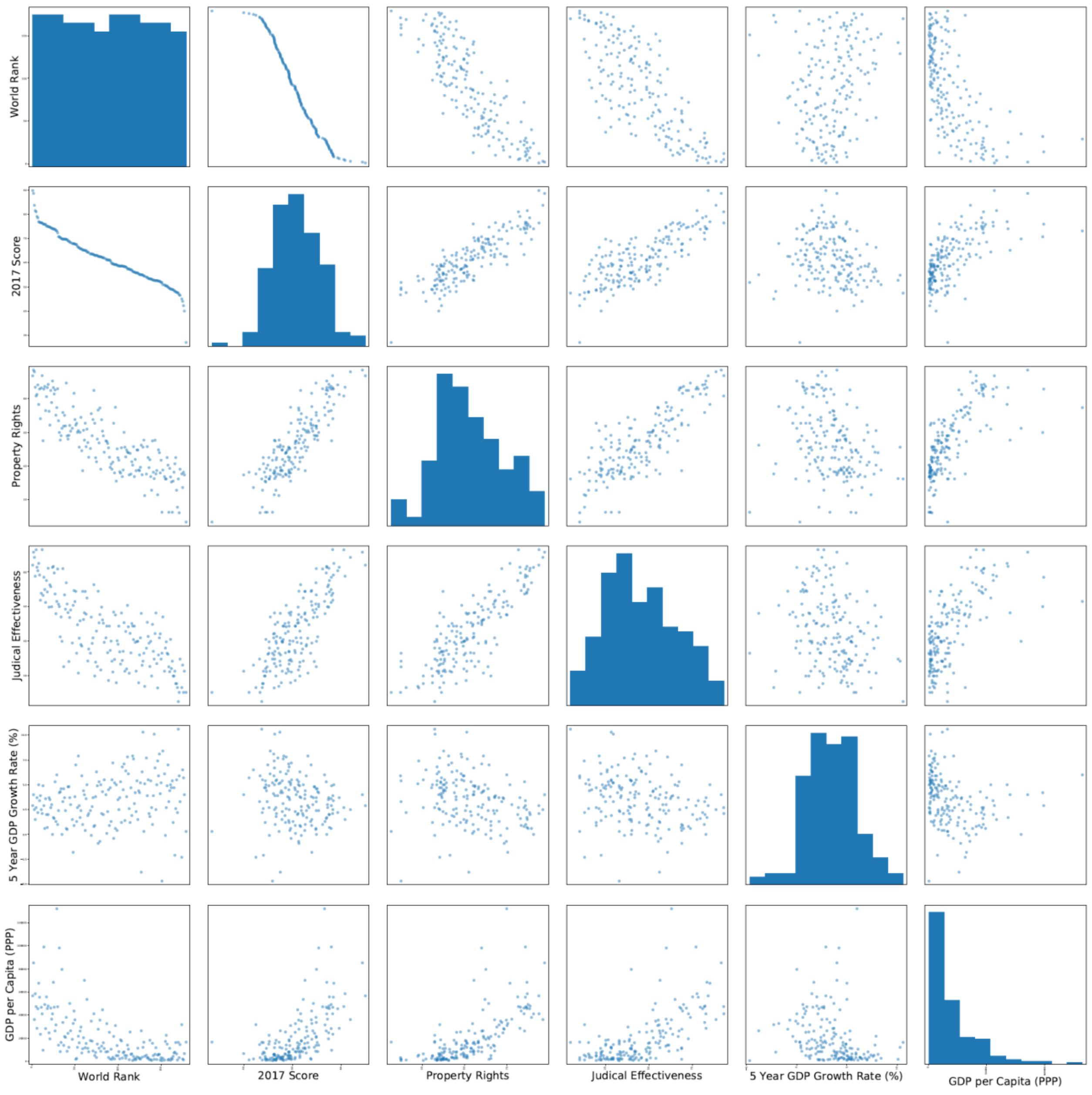


Figure 2.3

iii. Scatterplots and Distributions in a Single Figure

Finally, we create a figure that shows scatterplots of a list of selected variables as well as the distribution of each individual variable. Pandas already has a function built in for this. Due to the variance of figure size, we adjust the size of the text and scatterplot according to the dimensions of the figure.

```python
#econFreedomVisualizations.py
. . .

def formatted_scatter_matrix(data, pp):
    # Create a figure showing scatterplots given in scatter_cats
    fig_len = 15
    fig, ax = plt.subplots(figsize = ((fig_len, fig_len)))
    # Use fig_len to dictate fig_size, adjust size of font, size of dots, etc...
    num_vars = len(data.keys())
    fontsize = 65 / num_vars
    plt.rcParams.update({'font.size': fontsize})
    pd.plotting.scatter_matrix(data,alpha = .5, s = 108 / num_vars, ax=ax)
    # tight layout improves layout of text and plots in the figure
    plt.tight_layout()
    plt.show()
    pp.savefig(fig, bbox_inches = "tight")
    plt.close()


data = pd.read_csv("cleanedEconFreedomData.csv", index_col = ["Country Name"])
corr_data = pd.read_csv("econFreedomCorrMatrix.csv", index_col = [0])
# Save plots in a pdf using PdfPages
pp = PdfPages("Economic Freedom Plots.pdf")
# Set size of font used unless otherwise specified
plt.rcParams.update({"font.size": 26})
# select subset of variables to visualize in scatter plot
scatter_cats = ["World Rank",# "2017 Score", "Property Rights",
                "Judical Effectiveness", "5 Year GDP Growth Rate (%)",
                "GDP per Capita (PPP)"]
select_data = data[scatter_cats]
select_corr_data = corr_data.loc[scatter_cats][scatter_cats]
color_dim_scatter(select_data, pp)
corr_matrix_heatmap(select_corr_data, pp)
formatted_scatter_matrix(select_data, pp)
pp.close()
```



Exercises:

1. Find a dataset of your own choosing and import it as a dataframe using *pandas*. Clean the data set (i.e., drop missing values, transform non-numeric values using pd.to_numeric, etc…). Print resultant dataframe in the console.

2. Create a function that builds tables for variance, covariance, and correlation statistics for every variable in a dataframe. Use the function to create tables for a data set that you can use for a project.

Save each resultant dataframe as a csv named “dataVariance.csv”, dataCovariance.csv”, and “dataCorrelation.csv”.

3. In chapter 6 you created a series of scatter plots that integrate a 3rd variable by adjusting the color according to the value of that variable. Each of these plots is saved in a PDF. Create a function that performs these operations on all variables in a dataframe. (You may want to pass to the function a dataframe that selects from a limited number of keys in your dataset.) Be sure to pass at least two other objects to this function (e.g., fontsize, figsize, etc..).

In creating this function, remove any redundancies where the two plots share the same x and y variables (i.e., if key1 is on the horizontal axis and key2 on the vertical axis with key3 being represented by color, remove any graph where key 3 remains the same but key1 and key2 are switched.) Use this function to visualize the dataset that you chose for question #1.

4. Create a function that integrates the heatmap that you have created. Include an option for changing the title, fontsize of the title. Use the function to visualize a correlation matrix of the data that you chose for question #1.

7. Building an OLS Regression Model

Having built statistics functions, we are now ready to build a function for regression analysis. We will start by building the an regression. 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.

$$y = \alpha + \beta_1 x_1 + \mu$$

$$\beta_0 = \alpha + \mu$$

We merge the error term, which represents bias in the data, with alpha to yield the constant, β_0. This is necessary since OLS assumes an unbiased estimator where:

$$\sum_{i=0}^{j} e_i = 0$$

Each estimate of a point created from a particular observation takes the form.

$$y_i = \beta_0 + \beta_1 x_{1,i} + e_i$$

This can be generalized to include n exogenous, continuous variables:

$$y_i = \beta_0 + \sum_{j=1}^{n} \beta_j x_{i,j} + e_i$$

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 4.

To find the function that minimizes the sum squared errors, we will use matrix algebra, also known as linear algebra. For those unfamiliar, the next section uses the numpy library 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 a vector of beta values ($\boldsymbol{\beta}$), constants whose values represent estimates of the effect of variables in the set $\boldsymbol{X}$ on the selected endogenously generate variable $\boldsymbol{y}$**.** The matrix $\boldsymbol{X}$ also includes a vector of ones used to estimate the constant β_0.

$\boldsymbol{\beta} = (\boldsymbol{X'X})^{-1}\boldsymbol{X'Y}$

$\boldsymbol{Y}$ = Observations for Endogenous Variable
$\boldsymbol{X}$ = Observations for Exogenous Variables
$\boldsymbol{X'}$ = X-transpose
$(\boldsymbol{X'X})^{-1} = Inverse\ of\ \boldsymbol{X'X}$

Inverting a Matrix

In reviewing the linear equation for estimating $\boldsymbol{\beta},$ we confront two unique operations worth understanding. Included in these are some key concepts in linear algebra, including the identity matrix $\boldsymbol{I}$ and linear independence. The best way to understand these concepts is by working with some sample vectors. Consider the matrix $\boldsymbol{X}$ consisting of vectors $\boldsymbol{x_0}, \boldsymbol{x_1}, \dots, \boldsymbol{x_{n-1}}, \boldsymbol{x_n}$. We must check that these vectors are linearly independent. We do this by joining $\boldsymbol{X}$ with an identity matrix and thus create:

$$\boldsymbol{A} = [\boldsymbol{X}\ \boldsymbol{I}]$$

We can transform this such that:

$$\boldsymbol{AX^{-1}} = [\boldsymbol{X}\ \boldsymbol{I}]\boldsymbol{X^{-1}}$$

Since

$$\boldsymbol{AX^{-1}} = [\boldsymbol{I}\ \boldsymbol{X^{-1}}]$$

Let us solve for $\boldsymbol{AX^{-1}}$ using the following vectors for X.

$$\boldsymbol{X} = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{1} \\ \mathbf{4} & \mathbf{1} & \mathbf{5} \\ \mathbf{6} & \mathbf{8} & \mathbf{6} \end{bmatrix}$$

The identity matrix for a 3 x 3 matrix is:

$$\boldsymbol{I} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

We combine these

$$[\boldsymbol{X}\ \boldsymbol{I}\ ] = \begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{4} & \mathbf{1} & \mathbf{5} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{6} & \mathbf{8} & \mathbf{6} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

If we perform row operations on $\boldsymbol{A}$ to transform $\boldsymbol{X}$ in $[\boldsymbol{X}\ \boldsymbol{I}\ ]$ into $\boldsymbol{I}$, then we $\boldsymbol{I}$ will be transformed into $\boldsymbol{X^{-1}}$

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 4 & 1 & 5 & 0 & 1 & 0 \\ 6 & 8 & 6 & 0 & 0 & 1 \end{bmatrix}$$

r2 – 4r1

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & -7 & 1 & -4 & 1 & 0 \\ 6 & 8 & 6 & 0 & 0 & 1 \end{bmatrix}$$

r3 – 6r1

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & -7 & 1 & -4 & 1 & 0 \\ 0 & -4 & 0 & -6 & 0 & 1 \end{bmatrix}$$

r2 ⇔ r3

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & -4 & 0 & -6 & 0 & 1 \\ 0 & -7 & 1 & -4 & 1 & 0 \end{bmatrix}$$

r2 / -4

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 3/2 & 0 & -1/4 \\ 0 & -7 & 1 & -4 & 1 & 0 \end{bmatrix}$$

r3 + 7r2

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 3/2 & 0 & -1/4 \\ 0 & 0 & 1 & 13/2 & 1 & -7/4 \end{bmatrix}$$

r1 – 2r2 – r3

$$\begin{bmatrix} 1 & 0 & 0 & -17/2 & -1 & 9/4 \\ 0 & 1 & 0 & 3/2 & 0 & -1/4 \\ 0 & 0 & 1 & 13/2 & 1 & -7/4 \end{bmatrix}$$

$$\boldsymbol{IX^{-1}} = \begin{bmatrix} 1 & 0 & 0 & -8.5 & -1 & 2.25 \\ 0 & 1 & 0 & 1.5 & 0 & -.25 \\ 0 & 0 & 1 & 6.5 & 1 & -1.75 \end{bmatrix}$$

$$\boldsymbol{X^{-1}} = \begin{bmatrix} -8.5 & -1 & 2.25 \\ 1.5 & 0 & -.25 \\ 6.5 & 1 & -1.75 \end{bmatrix}$$

By transforming matrix ***X*** into matrix ***I*** we confirm that the vectors comprising ***X*** 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.

Linear Algebra in NumPy

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

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

Output:

```
Array 1:
[1 2 1]
Array 2:
[4 1 5]
Array 3:
[6 8 6]
```

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

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

x1 = np.matrix(x1)
x2 = np.matrix(x2)
x3 = np.matrix(x3)
print("Row Vector 1:",x1, sep="\n")
print("Row Vector 2:",x2, sep="\n")
print("Row Vector 3:",x3, sep="\n")
```

Output:

```
Array 1:
[1 2 1]
```

```
Array 2:
[4 1 5]
Array 3:
[6 8 6]
Row Vector 1:
[[1 2 1]]
Row Vector 2:
[[4 1 5]]
Row Vector 3:
[[6 8 6]]
```

Join them using the *concatenate()* function. We define *axis=0* to stack the from left to right.

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

x1 = np.matrix(x1)
x2 = np.matrix(x2)
x3 = np.matrix(x3)
print("Row Vector 1:",x1, sep="\n")
print("Row Vector 2:",x2, sep="\n")
print("Row Vector 3:",x3, sep="\n")

X = np.concatenate((x1,x2,x3), axis=0)
print("X:", X, sep="\n")
```

Output:

```
Array 1:
[1 2 1]
Array 2:
[4 1 5]
Array 3:
[6 8 6]
Row Vector 1:
[[1 2 1]]
Row Vector 2:
[[4 1 5]]
Row Vector 3:
[[6 8 6]]
X:
[[1 2 1]
 [4 1 5]
 [6 8 6]]
```

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

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

x1 = np.matrix(x1)
x2 = np.matrix(x2)
x3 = np.matrix(x3)
print("Row Vector 1:",x1, sep="\n")
print("Row Vector 2:",x2, sep="\n")
print("Row Vector 3:",x3, sep="\n")

X = np.concatenate((x1,x2,x3), axis=0)
X_inverse = X.getI()
print("X:", X, sep="\n")
print("X Inverse:", X_inverse, sep="\n")
```

Output:

```
Array 1:
[1 2 1]
Array 2:
[4 1 5]
Array 3:
[6 8 6]
Row Vector 1:
[[1 2 1]]
Row Vector 2:
[[4 1 5]]
Row Vector 3:
[[6 8 6]]
X:
[[1 2 1]
 [4 1 5]
 [6 8 6]]
X Inverse:
[[-8.5000000e+00 -1.0000000e+00  2.2500000e+00]
 [ 1.5000000e+00 -7.6861594e-17 -2.5000000e-01]
[ 6.5000000e+00  1.0000000e+00 -1.7500000e+00]]
```

These values are not rounded, so interpretation of the inverted matrix could be more intuitive. We use the *round()* method from the *numpy* module to round values to two places.

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

x1 = np.matrix(x1)
x2 = np.matrix(x2)
x3 = np.matrix(x3)
print("Row Vector 1:",x1, sep="\n")
print("Row Vector 2:",x2, sep="\n")
print("Row Vector 3:",x3, sep="\n")

X = np.concatenate((x1,x2,x3), axis=0)
X_inverse = np.round(X.getI(), 2)
print("X:", X, sep="\n")
print("X Inverse:", X_inverse, sep="\n")
```

Output:

```
Array 1:
[1 2 1]
Array 2:
[4 1 5]
Array 3:
[6 8 6]
Row Vector 1:
[[1 2 1]]
Row Vector 2:
[[4 1 5]]
Row Vector 3:
[[6 8 6]]
X:
[[1 2 1]
 [4 1 5]
 [6 8 6]]
X Inverse:
[[-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

$$\boldsymbol{\beta} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{Y}$$

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 saved in the previous chapter.

```
#econFreedomRegression
import pandas as pd

data = pd.DataFrame.from_csv("cleanedEconFreedomData.csv")
```

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.

In the console, enter the following:

```
data
```

Output:

```
                World Rank   ...     Public Debt (% of GDP)
Country Name                 ...
Afghanistan          163.0   ...              6.8
Albania              65.0    ...              71.9
Algeria              172.0   ...              8.7
Angola               165.0   ...              62.3
Argentina            156.0   ...              56.5
Armenia              33.0    ...              46.6
Australia           5.0      ...              36.8
Austria              30.0    ...              86.2
Azerbaijan           68.0    ...              36.1
Bahamas              90.0    ...              65.7
Bahrain              44.0    ...              63.3
Bangladesh           128.0   ...              34.0
Barbados             130.0   ...              103.0
Belarus              104.0   ...              59.9
Belgium              49.0    ...              106.3
Belize               101.0   ...              76.3
Benin                96.0    ...              37.5
Bhutan               107.0   ...              115.7
```

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.

```python
#regression.py
import pandas as pd
import copy
from stats import *

class Regression:
    def __init__(self):
        self.stats = Stats()

    def OLS(self, reg_name, data, y_name, beta_names, min_val = 0,
                max_val = None, constant = True):
        self.min_val = min_val
        if max_val != None:
            self.max_val = max_val
        else:
            self.max_val = len(data)
        self.reg_name = reg_name
        self.y_name = y_name
        self.beta_names = copy(beta_names)
        self.data = data.copy()
        if constant:
            self.add_constant()

    def add_constant(self):
        self.data["Constant"] = 1
        self.beta_names.append("Constant")
```

We start by importing pandas, and the stats py file that we have already saved in the same folder. We create two methods inside of the *Regression* class. First is the *__init__* method. This will create an instance of *Stats* that will be called later. Second is the *regress* method. This is our primary method, from which all the necessary steps for preparing data and running a regression will be called.

The *OLS* method passes several objects. First is reg_name, which is a string that be included in the regression summary ouput. Data is the pandas data frame used for the regression. Next are the names of the variables we wish to regress: *y_name* is the name of the dependent variable and *x_names* is a list that includes the names of variables that we wish to regress. *min_val* and *max_val* are the starting and ending index values for the regression.

OLS includes the option to include a constant. If *constant = True*, then a column of ones is added to the data. This column will be used to estimate a constant that determines at what value the fitted line or curve crosses the y-axis. Increase or decrease in this value shift the line up or down, respectively.

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

```python
#econFreedomRegression
import pandas as pd
import regression

data = pd.DataFrame.from_csv("cleanedEconFreedomData.csv")
reg = regression.Regression()
```

Now that we have created an instance of the *Regression* class, we can use the stats
Output:

> <module 'regression' from 'E:\\For NDSU\\Teaching\\Spring 2019\\ECON 499 696\\Class Examples\\Section 7\\regression.py'>

A standard OLS regression assumes that the equation it is estimating will include a constant. We must therefore include a the option to include a constant, or not, in the estimation. 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”.

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.

Selecting Variables

We may ask how different types of freedom toend to affect prosperity within a nation. The Heritage Index of Economic Freedom provides different measures to rate freedom and the rule of law within a country. We use these to predict GDP per capita in each country. Below, is the script that will be used to run a regression. One indicator of the quality of explanatory power provided by an exogenous variable is it’s ability to maintain a steady value in terms of its estimated effect on the endogenous variable as well as its statistical significance. We will return to this discussion once the regression class is completed.

```python
#econFreedomRegression
import pandas as pd
import regression

data = pd.read_csv("cleanedEconFreedomData.csv")
reg = regression.Regression()

y_var = ["GDP per Capita (PPP)"]
x_vars = ["Trade Freedom", "Property Rights", "Judical Effectiveness",
          "Fiscal Health", "Investment Freedom ", "Financial Freedom",
          "Inflation (%)", "Public Debt (% of GDP)"]

reg.OLS("GDP Per Capita", data, y_var, x_vars)
```

We will run the above script to check our progress as we develop the *Regression* class. For now, execute the script. Then, in the console, use the following to print the data with the constant:

```
reg.data["Constant"]
```

Output:

```
Country Name
Afghanistan         1
Albania             1
Algeria             1
Angola              1
Argentina           1
Armenia             1
Australia           1
Austria             1
…
Zambia              1
Zimbabwe            1
Brunei Darussalam   1
Name: Constant, Length: 172, 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, includnig the vector of constants that we created. We must also create a transpose of the X matrix and the inverse of X'X. With these, we will be able to implement an OLS regression.

```
#regression.py
import pandas as pd
import  copy
import numpy as np
from stats import *
class Regression:
    def __init__(self):
        self.stats = Stats()

    def OLS(self, reg_name, data, y_name, beta_names, min_val = 0,
                max_val = None, constant = True):
        . . .
        self.build_matrices()
        self.estimate_betas_and_yhat()
        . . .
    def build_matrices(self):
        # Transform dataframes to matrices
        self.y = np.matrix(self.data[self.y_name][self.min_val:self.max_val])
        # create a k X n nested list containg vectors for each exogenous var
        self.X = np.matrix(self.data[self.beta_names])
        self.X_transpose = np.matrix(self.X).getT()
        # (X'X)**-1
        X_transp_X = np.matmul(self.X_transpose, self.X)
        self.X_transp_X_inv = X_transp_X.getI()
        # X'y
        self.X_transp_y = np.matmul(self.X_transpose, self.y)

    def estimate_betas_and_yhat(self):
        # betas = (X'X)**-1 * X'y
        self.betas = np.matmul(self.X_transp_X_inv, self.X_transp_y)
        # y-hat = X * betas
        self.y_hat = np.matmul(self.X, self.betas)
        # Create a column that holds y-hat values
        self.data[self.y_name[0] + " estimator"] = \
            [i.item(0) for i in self.y_hat]
        # create a table that holds the estimated coefficient
        # this will also be used to store SEs, t-stats,and p-values
        self.estimates = pd.DataFrame(self.betas, index = self.beta_names,
                                      columns = ["Coefficient"])
        # identify y variable in index
        self.estimates.index.name = "y = " + self.y_name[0]
```

From the *econFreedomRegression.py* file, lets execute the *regress* function that we have extended. Executing it will generate the data frame of coefficient estimates. In the console enter

```
reg.estimates
```

Output:

```
y = GDP per Capita (PPP)        Coefficient
Trade Freedom                     -3.076745
Property Rights                  515.155805
Judical Effectiveness            233.867234
Fiscal Health                     11.250231
Investment Freedom               -72.596777
Financial Freedom                222.833802
Inflation (%)                    215.060317
Public Debt (% of GDP)           -79.676089
Constant                           -0.01511
```

We have calculated beta values for each independent 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 measured the statistical 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 $\hat{y}$. We will refer to individual predicted values as $\hat{y_i}$. We will also refer to the mean value of observations of our dependent variable as $\bar{y}$ and individual observed values of our dependent variable as y_i. 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 y, the observed y, and the mean y, we will estimate the standard error for each coefficient and other values that estimate convey the significance of the estimation.

We define these values as follows:

$$SSR = \sum_{i}^{n} (\hat{y_i} - \bar{y})^2$$

$$SSE = \sum_{i}^{n} (y_i - \hat{y_i})^2$$

$$SST = \sum_{i}^{n} (y_i - \bar{y})^2$$

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:

$$SST = SSR + SSE$$

The script below will estimate these statistics. It calls the *sum_square_stats method* from the which is passed in the *calculate_regression_stats* method.

```
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
. . .

    def calculate_regression_stats(self):
        self.sum_square_stats()

    def sum_square_stats(self):
        ssr_list = []
        sse_list = []
        sst_list = []
        mean_y = self.stats.mean(self.y).item(0)
        for i in range(len(self.y)):
            # ssr is sum of squared distances between the estimated y values
            # (y-hat) and the average of y values (y-bar)
            yhat_i = self.y_hat[i]
            y_i = self.y[i]
            r = yhat_i - mean_y
            e = y_i - yhat_i
            t = y_i - mean_y
            ssr_list.append((r) ** 2)
            sse_list.append((e) ** 2)
            sst_list.append((t) ** 2)

        # call item - call value instead of matrix
        self.ssr = self.stats.total(ssr_list).item(0)
        self.sse = self.stats.total(sse_list).item(0)
        self.sst = self.stats.total(sst_list).item(0)
```

The elements of the lists created are actually matrices, so we select the element in the matrix by calling *.item(0)* after summing each list with *total()*.

Now, the regression method will allow us to call the SSR, SSE, and SST values. These will be used to calculate the measures commonly associated with a regression such as r-squared and estimator variance. In the console enter:

```
print(reg.ssr, reg.sse, reg.sst)
```

Output:

```
42755533480.27381, 37376553427.92663, 80132086908.20023
```

With the sum of squared errors calculated, the next step is to calculate the estimator variance and use this to construct the covariance matrix. The covariance matrix is used to derive the standard errors and related statistics for each estimated coefficient.

We estimate the variance of the error term of the estimator for the dependent variable.

$$\sigma^2 = \frac{SSE}{n-k}$$

$$n = number\ of\ obvservations$$
$$k = number\ of\ independent\ variables$$

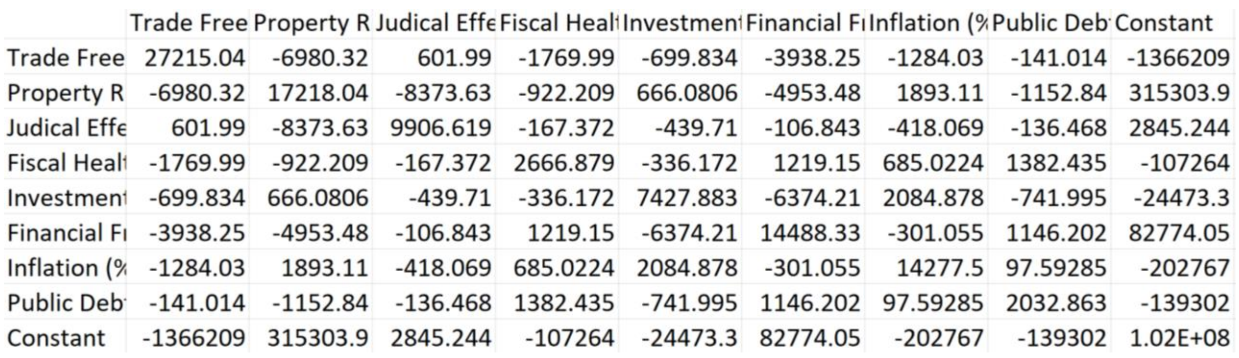
An increase in the number of exogenous variables tends ot increase the fit of a model. By dividing the SSE by degrees of freedom, $(n-k)$, improvements in fit that result from increases in the number of variables are offset in part by a reduction in degrees of freedom.

```
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
. . .
```

```python
    def calculate_regression_stats(self):
        self.sum_square_stats()
        self.calculate_degrees_of_freedom()
        self.calculate_estimator_variance()
        self.calculate_covariance_matrix()
 . . .
    def calculate_degrees_of_freedom(self):
        # Degrees of freedom compares the number of observations to the number
        # of exogenous variables used to form the prediction
        self.lost_degrees_of_freedom = len(self.estimates)
        self.num_obs = self.max_val + 1 - self.min_val
        self.degrees_of_freedom = self.num_obs - self.lost_degrees_of_freedom

    def calculate_estimator_variance(self):
        # estimator variance is the sse normalized by the degrees of freedom
        # thus, estimator variance increases as the number of exogenous
        # variables used in estimation increases(i.e., as degrees of freedom
        # fall)
        self.estimator_variance = self.sse / self.degrees_of_freedom

    def calculate_covariance_matrix(self):
        # Covariance matrix will be used to estimate standard errors for
        # each coefficient.
        # estimator variance * (X'X)**-1
        self.cov_matrix = float(self.estimator_variance) * self.X_transp_X_inv
        self.cov_matrix = pd.DataFrame(self.cov_matrix,
                                       columns = self.beta_names,
                                       index = self.beta_names)
```

In the final method, *calculate_covariance_matrix*, the estiamtor variance is used to calculate the estimator covariance matrix. We will view this table by creating a csv. Enter the following command into the console:

```
reg.cov_matrix.to_csv(“regCovMatrix.csv”)
```



	Trade Free	Property R	Judical Effe	Fiscal Heal	Investment	Financial Fi	Inflation (%	Public Deb	Constant
Trade Free	27215.04	-6980.32	601.99	-1769.99	-699.834	-3938.25	-1284.03	-141.014	-1366209
Property R	-6980.32	17218.04	-8373.63	-922.209	666.0806	-4953.48	1893.11	-1152.84	315303.9
Judical Effe	601.99	-8373.63	9906.619	-167.372	-439.71	-106.843	-418.069	-136.468	2845.244
Fiscal Heal	-1769.99	-922.209	-167.372	2666.879	-336.172	1219.15	685.0224	1382.435	-107264
Investment	-699.834	666.0806	-439.71	-336.172	7427.883	-6374.21	2084.878	-741.995	-24473.3
Financial Fi	-3938.25	-4953.48	-106.843	1219.15	-6374.21	14488.33	-301.055	1146.202	82774.05
Inflation (%	-1284.03	1893.11	-418.069	685.0224	2084.878	-301.055	14277.5	97.59285	-202767
Public Deb	-141.014	-1152.84	-136.468	1382.435	-741.995	1146.202	97.59285	2032.863	-139302
Constant	-1366209	315303.9	2845.244	-107264	-24473.3	82774.05	-202767	-139302	1.02E+08

The diagonals of this table represent the standard errors for each variable. The standard what we will use to calculate t-statistics and p-values. The t-statisitic of a coefficient is found by comparing the size of the estimated coefficient to its standard error:

$$tstat_{\beta_i} = \frac{\beta_i}{SE_{\beta_i}}$$

The larger the coefficient compared to the error, the more reliable is the statistic, as implied by a large t-stat. We draw the p-value associated with a particular t-stat from a table in light of the degrees of freedom associated with the regression. The p-values provides a rating of the estimate in light of the t-stat together with the number of degrees of freedom.

```python
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
from scipy.stats import t, f
. . .
    def calculate_regression_stats(self):
        self.sum_square_stats()
        self.calculate_degrees_of_freedom()
        self.calculate_estimator_variance()
        self.calculate_covariance_matrix()
        self.calculate_t_p_error_stats()
  . . .

    def calculate_t_p_error_stats(self):
        ratings = [.05, .01, .001]
        results = self.estimates
        stat_sig_names = ["SE", "t-stat", "p-value"]
        # create space in data frame for SE, t, and p
        for stat_name in stat_sig_names:
            results[stat_name] = np.nan
        # generate statistic for each variable
        for var in self.beta_names:
            # SE ** 2 of coefficient is found in the diagonal of cov_matrix
            results.loc[var]["SE"] = self.cov_matrix[var][var] ** (1/2)

            # t-stat = Coef / SE
            results.loc[var]["t-stat"] = \
                results["Coefficient"][var] / results["SE"][var]
            # p-values is estimated using a table that transforms t-value in
            # light of degrees of freedom
            results.loc[var]["p-value"] = np.round(t.sf(np.abs(results.\
                           loc[var]["t-stat"]), self.degrees_of_freedom + 1) * 2, 5)
        # values for significances will be blank unless p-values < .05
        # pandas does not allow np.nan values or default blank strings to
        # be replaced x-post
        significance = ["" for i in range(len(self.beta_names))]
        for i in range(len(self.beta_names)):
            var = self.beta_names[i]
            for val in ratings:
                if results.loc[var]["p-value"] < val:
                    significance[i] = significance[i]  + "*"
        results["signficance"] = significance
```

The standard errors, t-statistics and p-values are saved in the same dataframe as the coefficient estimates. Call them with the command:

```
reg.estimates.to_csv("estimates.csv")
```

y = GDP per Capita (PPP)	**Coefficient**	**SE**	**t-stat**	**p-value**	**significance**
Trade Freedom	-3.076744557	164.9698234	-0.018650348	0.98514	
Property Rights	515.1558047	131.2175248	3.92596801	0.00013	***
Judical Effectiveness	233.8672338	99.53199811	2.349668833	0.01997	*
Fiscal Health	11.25023072	51.64183358	0.217851109	0.82781	
Investment Freedom	-72.59677657	86.18516476	-0.842334951	0.40082	
Financial Freedom	222.8338025	120.3674707	1.851279263	0.06592	
Inflation (%)	215.0603169	119.4884914	1.79984126	0.07371	
Public Debt (% of GDP)	-79.67608949	45.08728256	-1.767152176	0.07905	
Constant	-22337.14233	10090.24194	-2.213737041	0.02822	*

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, R^2, and F-stat.

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.

$$rootMSE = \sqrt{\sigma^2}$$

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 R^2 value. This value indicates the explanator power of the regression

$$R^2 = \frac{SSR}{SST}$$

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:

$$F = \frac{\frac{SST - SSE}{K - 1}}{\frac{SSE}{N - K}}$$

The F-statistic tests the likelihood of whether or not the values of our estimated parameters are all zero:

$$\beta_1 = \beta_2 = \cdots = \beta_{n-1} = \beta_n = 0$$

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.

```
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
from scipy.stats import t
. . .
    def calculate_regression_stats(self):
        self.sum_square_stats()
        self.calculate_degrees_of_freedom()
        self.calculate_estimator_variance()
        self.calculate_covariance_matrix()
        self.calculate_t_p_error_stats()
        self.calculate_root_MSE()
        self.calculate_rsquared()
        self.calculate_fstat()
. . .

            def calculate_root_MSE(self):
                self.root_mse = self.estimator_variance ** (1/2)

            def calculate_rsquared(self):
                self.r_sq = self.ssr / self.sst

            def calculate_fstat(self):
                self.f_stat = (self.sst - self.sse) / (self.lost_degrees_of_freedom\
                              - 1) / self.estimator_variance
```

Now that all of the regression statistics have been gathered, the only step left is to place the remaining statistics into a single table.

```
def calculate_regression_stats(self):
        self.sum_square_stats()
        self.calculate_degrees_of_freedom()
        self.calculate_estimator_variance()
        self.calculate_covariance_matrix()
        self.calculate_t_p_error_stats()
        self.calculate_MSE()
        self.calculate_rsquared()
        self.calculate_fstat()
        self.build_stats_DF()
. . .
    def build_stats_DF(self):
        stats_dict = {"r**2":[self.r_sq],
                      "f-stat":[self.f_stat],
                      "Est Var":[self.estimator_variance],
                      "rootMSE":[self.root_mse],
                      "SSE":[self.sse],
                      "SSR":[self.ssr],
                      "SST":[self.sst],}
                      "Obs.":[int(self.num_obs)],
                      "DOF":[int(self.degrees_of_freedom)]}
        self.stats_DF = pd.DataFrame(stats_dict)
        self.stats_DF = self.stats_DF.rename(index={0:"Estimation Statistics"})
        self.stats_DF = self.stats_DF.T
```

Be sure to call the function at the end of the *calculate_regression_stats()* method:

Save *stats_DF* as a csv with the command:

```
reg.stats_DF.to_csv("reg_stats.csv")
```

This yields the table:

	Estimation Statistics
r2**	0.53356321
f-stat	23.45022095
Est Var	227905813.6
MSE	15096.54972
SSE	37376553428
SSR	42755533480
SST	80132086908
Obs.	173
DOF	164

You have successfully created a program that runs completes OLS regression and organizes statistics from this regression!

Tests and Adjustments

In addition to the essential elements that you have included in the regression method, some other evaluative critieria is in order. We will include calculate the adjusted r-squared as well as joint f-tests. The first of these is used to offset the increase in the r-squared value that otherwise occurs when exogenous variables are added to a regression. It is possible that the addition of irrevelevant variables appear to increase goodness-of-fit.

For similar purposes, we will include the joint F-test. This compares the explanatory power of two regressions, revealing whether or not the inclusion of additional variables is actually improving explanatory power of the regression.

Adjusted R-Squared

Although the R^2 is a useful measure to understand the quality of the explanation provided by the selected exogenous variables. Recall that:

$$R^2 = \frac{SSR}{SST}$$

$$R^2 = 1 - \frac{SSE}{SST}$$

An as more explanatory variables are added, the SSE falls and the R^2 rises as a result. The adjusted R^2 accounts for a decrease in degrees of freedom that occur with an increase in the number of explanatory variables. It takes the form:

$$\boldsymbol{R^2} = 1 - \frac{(\frac{SSE}{n-k})}{(\frac{SST}{n-1})}$$

Notice that as the degrees of freedom decrease, the numerator necessarily decreases as well. Although it is not always appropriate to use the adjusted R^2, it is often useful to help gauge whether or not a marginal addition of a variable improves explanatory power of a regression.

```
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
from scipy.stats import t
. . .
. . .
    def calculate_rsquared(self):
        self.r_sq = self.ssr / self.sst
        self.adj_r_sq = 1 - self.sse / self.degrees_of_freedom / (self.sst\
                             / (self.num_obs - 1))

     def calculate_fstat(self):
        self.f_stat = (self.sst - self.sse) / (self.lost_degrees_of_freedom\
                       - 1) / self.estimator_variance

     def build_stats_DF(self):
        stats_dict = {"r**2": [self.r_sq],
                      "Adj. r**2": [self.adj_r_sq],
                      "f-stat": [self.f_stat],
                      "EST Var": [self.estimator_variance],
                      "MSE": [self.mse],
                      "SSE": [self.sse],
                        "SSR": [self.ssr],
                        "SST": [self.sst]}
        self.stats_DF = pd.DataFrame(stats_dict)
        self.stats_DF = self.stats_DF.rename(index={0:"Estimation Statistics"})
        self.stats_DF = self.stats_DF.T
```

Joint F-test

Just as the adjusted r-squared allows for a more effective comparison of regressions that have varying numbers of variables, so too does the joint f-test. In order to compare regressions, we must save results from at least two compareable regression. To save results, we create a dictionary name *reg_history* and save this using the method, *save_output*.

```
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
from scipy.stats import t,

class Regression:
    def __init__(self):
        self.stats = Stats()
        self.reg_history = {}
    . . .
        self.calculate_regression_stats()
        self.save_output()
. . .
    def save_output(self):
        self.reg_history[self.reg_name] = {"Reg Stats": self.stats_DF.copy(),
                        "Estimates": self.estimates.copy(),
                        "Cov Matrix":self.cov_matrix.copy(),
                        "Data":self.data.copy()}
```

By saving the regression statistics, estimates, and covariance matrix in a dictionary with a unique key that is the string passed as *self.reg_name* for the regression, results from multiple regression can be called. This is required for running a joint_f_test, which is supposed to estimate whether or not the addition of an exogenous variable in a regression actually improves the explantory power of the regression.

The joint f-test compares a restricted and unrestricted regression. The unrestricted regression uses the same exogenous variables as the restricted regression, and adds at least one more exogenous variable to be used to estimate values of y. The joint f-test checks whether, the betas values of the exogenous variables included in the unretricted regression are different than zero.

To check, we calculate the joint F-statistic:

$$F = \frac{\frac{SSE_R - SSE_U}{k_R - k_U}}{\frac{SSE_U}{n - k_U}}$$

If the p-values associated with this f-statistic indicates statistical significance, then at least one of the additional variables improve the explanatory power of the regression.

```python

    def joint_f_test(self, reg1_name, reg2_name):
        # identify data for each regression
        reg1 = self.reg_history[reg1_name]
        reg2 = self.reg_history[reg2_name]
        # identify beta estimates for each regression to draw variables
        reg1_estimates = reg1["Estimates"]
        reg2_estimates = reg2["Estimates"]
        # name of y_var is saved as estimates index name
        reg1_y_name = reg1_estimates.index.name
        reg2_y_name = reg2_estimates.index.name
        num_obs1 = reg1["Reg Stats"].loc["Obs."][0]
        num_obs2 = reg2["Reg Stats"].loc["Obs."][0]
        # check that the f-stat is measuring restriction, not for diff data sets
        if num_obs1 != num_obs2:
            self.joint_f_error()
        if reg1_y_name == reg2_y_name:
            restr_reg = reg1 if \
                len(reg1_estimates.index) < len(reg2_estimates.index) else reg2
            unrestr_reg = reg2 if restr_reg is reg1 else reg1
            restr_var_names = restr_reg["Estimates"].index
            unrestr_var_names = unrestr_reg["Estimates"].index
        # identify statistics for each regression
        restr_reg = restr_reg if False not in \
                [key in unrestr_var_names for key in restr_var_names] else None
        if restr_reg == None:
            self.joint_f_error()
        else:
            sser = restr_reg["Reg Stats"].loc["SSE"][0]
            sseu = unrestr_reg["Reg Stats"].loc["SSE"][0]
            dofr = restr_reg["Reg Stats"].loc["DOF"][0]
            dofu = unrestr_reg["Reg Stats"].loc["DOF"][0]
            dfn = dofr - dofu
            dfd = dofu - 1
            f_stat = ((sser - sseu) / (dfn)) / (sseu / (dfd))
            f_crit_val = 1 - f.cdf(f_stat,dfn = dfn, dfd = dfd)
            #make dictionary?
            f_test_label = ""
            for key in unrestr_var_names:
                if key not in restr_var_names:
                    f_test_label = f_test_label + str(key) + " = "
            f_test_label = f_test_label + "0"
            res_dict = {"f-stat":[f_stat],
                        "p-value":[f_crit_val],
                        "dfn":[dfn],
                        "dfd":[dfd]}
            res_DF = pd.DataFrame(res_dict)
            res_DF = res_DF.rename(index={0:""})
            res_DF = res_DF.T
            res_DF.index.name = f_test_label

            return res_DF

    def joint_f_error(self):
            print("Regressions not comparable for joint F-test")
            return None
```

Call the Joint F-Test

Having constructed a method to run the joint F-test, we first need to define the restricted and unrestricted regressoin. The restricted regression will have only two variables: “Judical Effectiveness” and “Property Rights”. These two variables are the only exogenous variables from the earlier regression that were significant. We will compare the regression that we ran, the unrestricted regression, with this restricted regression. Name each regression accordingly, then call the joint_f_test using the names for each regression. The program is distinguish automoticallly which is restricted and which is unrestricted. Since the results of the test are saved in a dataframe, we can save the results as a csv.

```python
#restrictedEconFreedomRegression
import pandas as pd
import regression
import statsmodels.api as sm

data = pd.read_csv("cleanedEconFreedomData.csv")
reg = regression.Regression()

y_var = ["GDP per Capita (PPP)"]
x_vars_unrestricted = ["Judical Effectiveness", "Property Rights",
          "Fiscal Health", "Investment Freedom ", "Financial Freedom",
          "Inflation (%)", "Public Debt (% of GDP)"]
x_vars_restricted = ["Judical Effectiveness", "Property Rights"]
reg.OLS("GDP Per Capita Restricted", data, y_var, x_vars_restricted)
reg.OLS("GDP Per Capita Unrestricted", data, y_var, x_vars_unrestricted)

joint_f_test = reg.joint_f_test("GDP Per Capita Unrestricted",
                                "GDP Per Capita Restricted")
joint_f_test.to_csv("Joint F_test; y = " + reg.y_name[0] + "; " +\
                    joint_f_test.index.name + ".csv")
```

The results are saved in a csv that denotes the constraint tested by the f-test:

Fiscal Health = Investment Freedom = Financial Freedom = Inflation (%) = Public Debt (% of GDP) = 0	
f-stat	2.783055
p-value	0.019294
dfn	5
dfd	164

Visualizing Regression Results

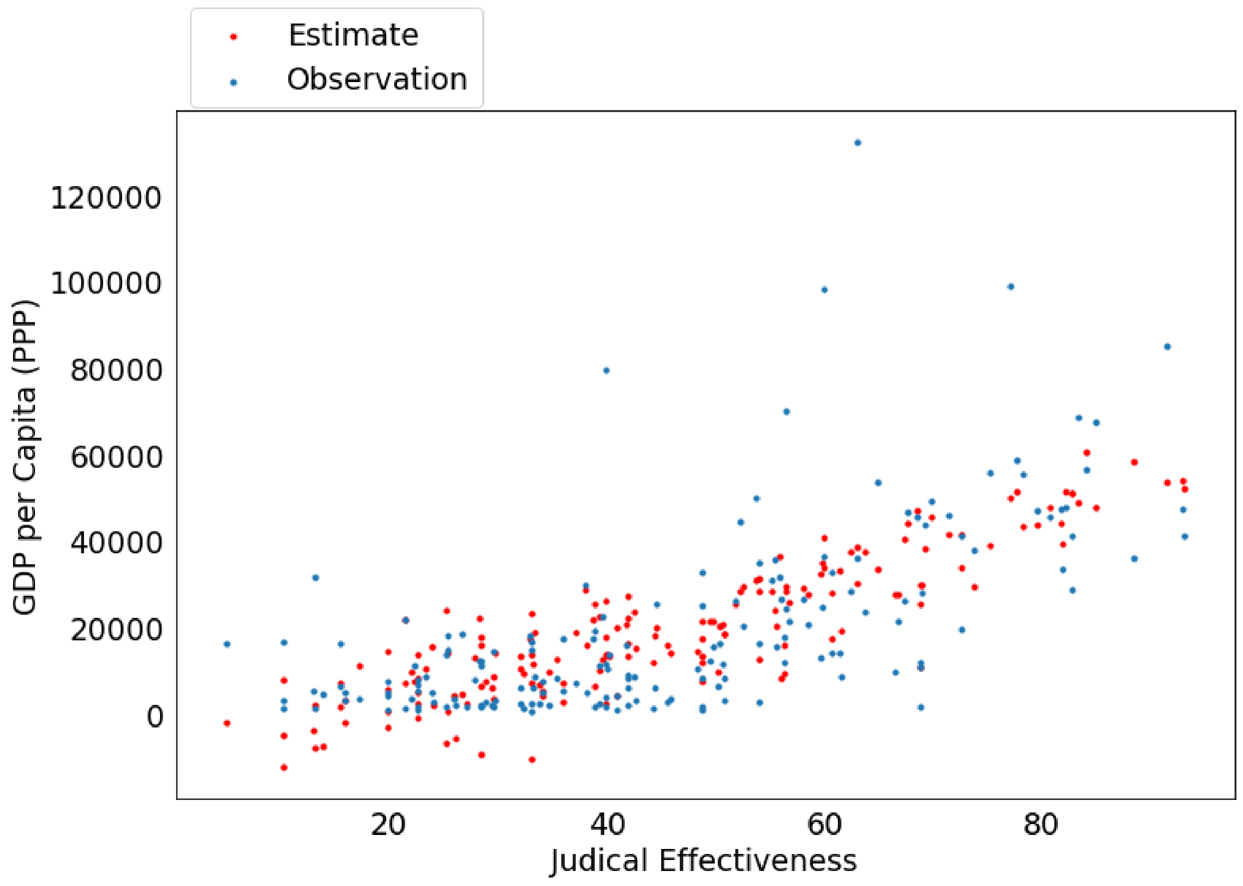
To get the most value out of a regression function requires not only tables for statistics. Visualizations that compare observations to estimates form the regression are a powerful means of presenting results. Since we have already generated estimations for y_hat, this simply requires the plotting of y values against values of exogenous variables. On the same plot, do the same for estimator (yhat) values.

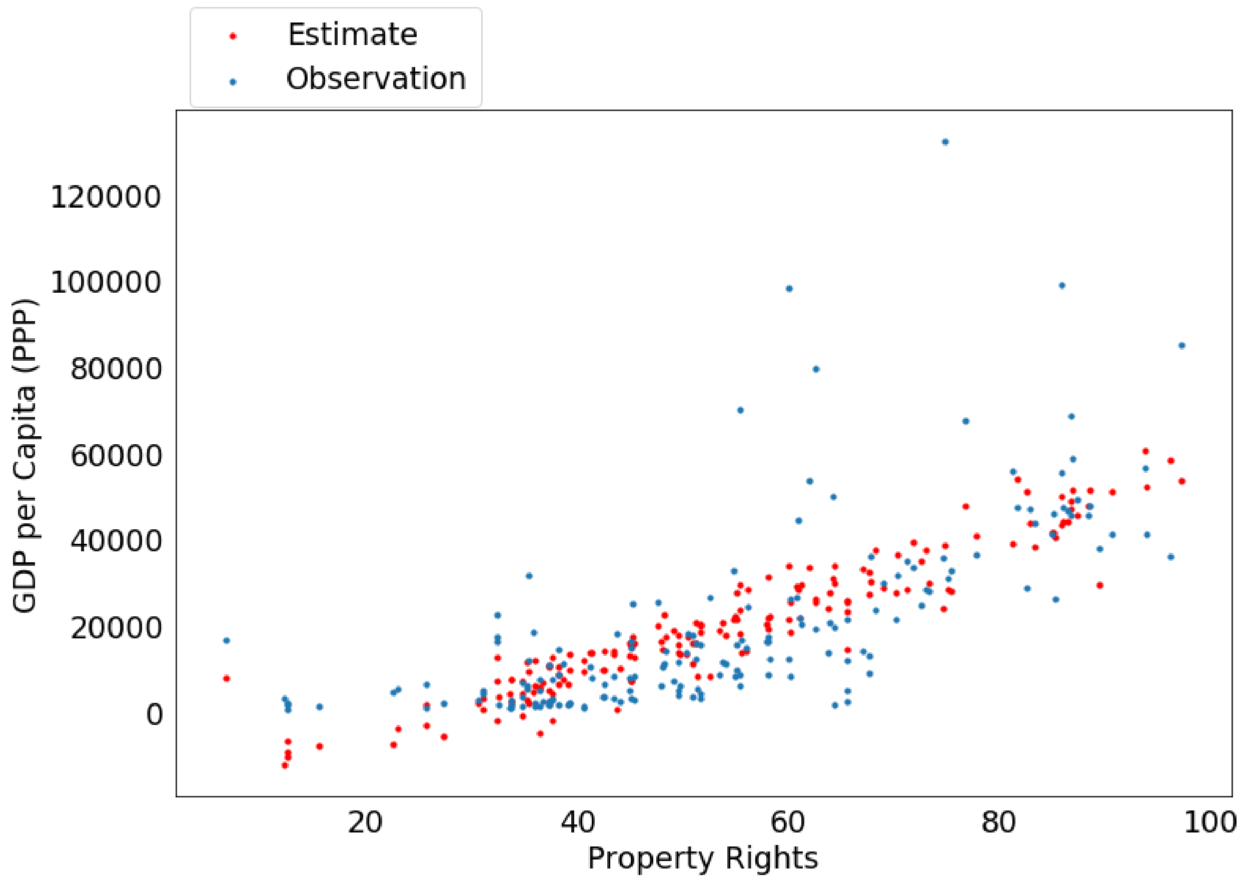
In the next section we will use a for loop to compare the predicted values of the y variable with observed values in a scatter plot. Each plot will include the y-values on the veritcal axis and the values of an exogenous variable on the y axis.

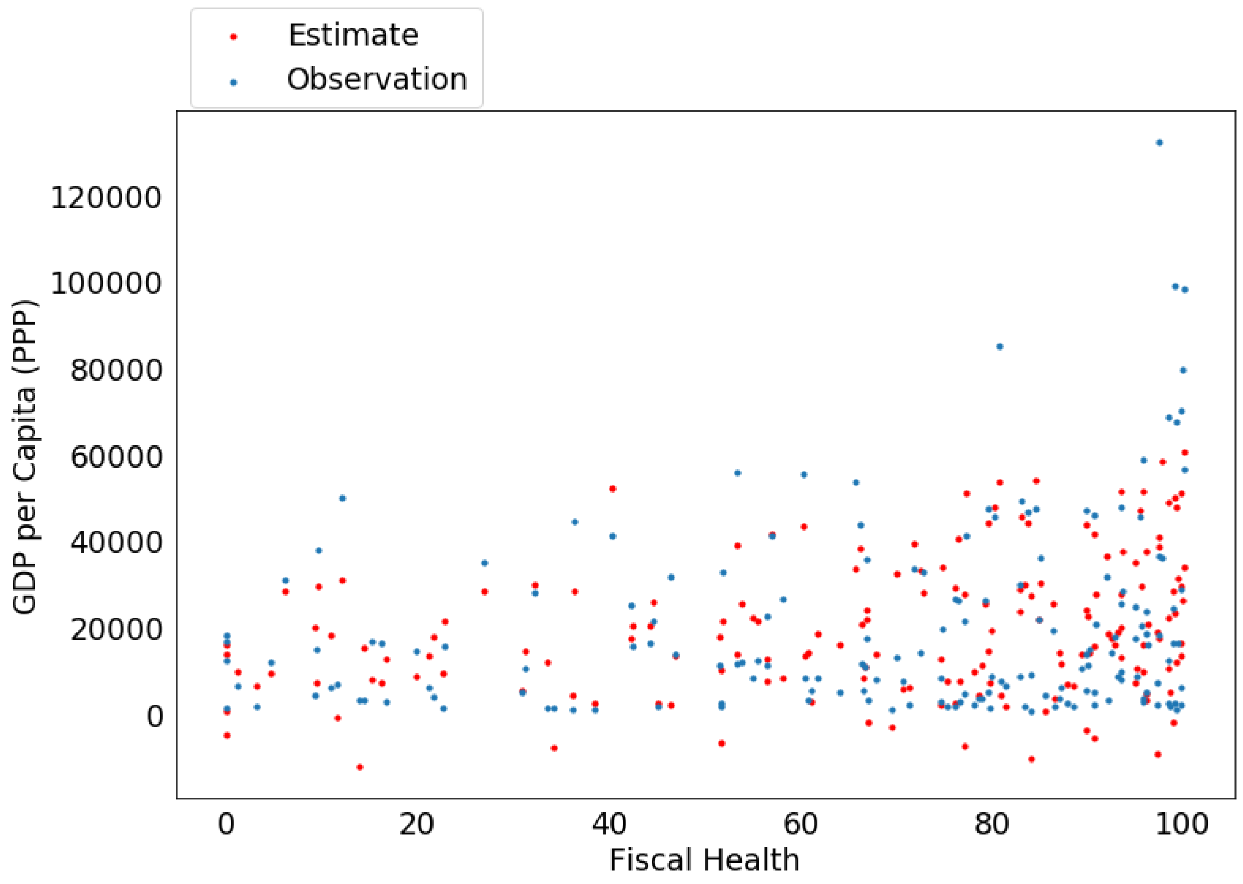
```python
#econFreedomRegression
import pandas as pd
import regression
import statsmodels.api as sm
import matplotlib.pyplot as plt

def plot_scatter_with_estimator(data, x_vars, y_var):
    # set default font size
    plt.rcParams.update({"font.size": 19})
    # use a for loop to call each exogenous variable
    y = y_var[0]
    for x in x_vars:
        # prepare a figure with the predictor. We will use ax to specify that
        # the plots are in the same figure
        fig, ax = plt.subplots(figsize = (12, 8))
        # labels will be in a legend
        y_label1 = "Estimate"
        y_label2 = "Observation"
        # plot the estimated value
        data.plot.scatter(x = x, y = y + " estimator", ax = ax, c = "r",
                          s = 10, label = y_label1, legend = False)
        # erase the y-axis label so that "estimator" is not present
        # the y-label will reappear when the observations are plotted
        plt.ylabel("")
        data.plot.scatter(x = x, y = y, ax = ax, s = 10, label = y_label2,
                          legend = False)
        # call the legend, place atop the image on the left
        # bbox_to_anchor used to specify exact placement of label
        plt.legend(loc = "upper left", labels = [y_label1, y_label2],
                   bbox_to_anchor = (0, 1.17))
        # remove lines marking units on the axis
        ax.xaxis.set_ticks_position('none')
        ax.yaxis.set_ticks_position('none')
        plt.show()
        plt.close()

data = pd.read_csv("cleanedEconFreedomData.csv")
reg = regression.Regression()

y_var = ["GDP per Capita (PPP)"]
x_vars = ["Judical Effectiveness", "Property Rights",
          "Fiscal Health", "Investment Freedom ", "Financial Freedom",
          "Inflation (%)", "Public Debt (% of GDP)"]
reg.OLS("GDP Per Capita Restricted", data, y_var, x_vars)
plot_scatter_with_estimator(reg.data, x_vars, y_var)
```

The visualization generated by this script allow for a comparison of estimates of y-variable generated from a set of observed values to the actual values that were part of the observations.

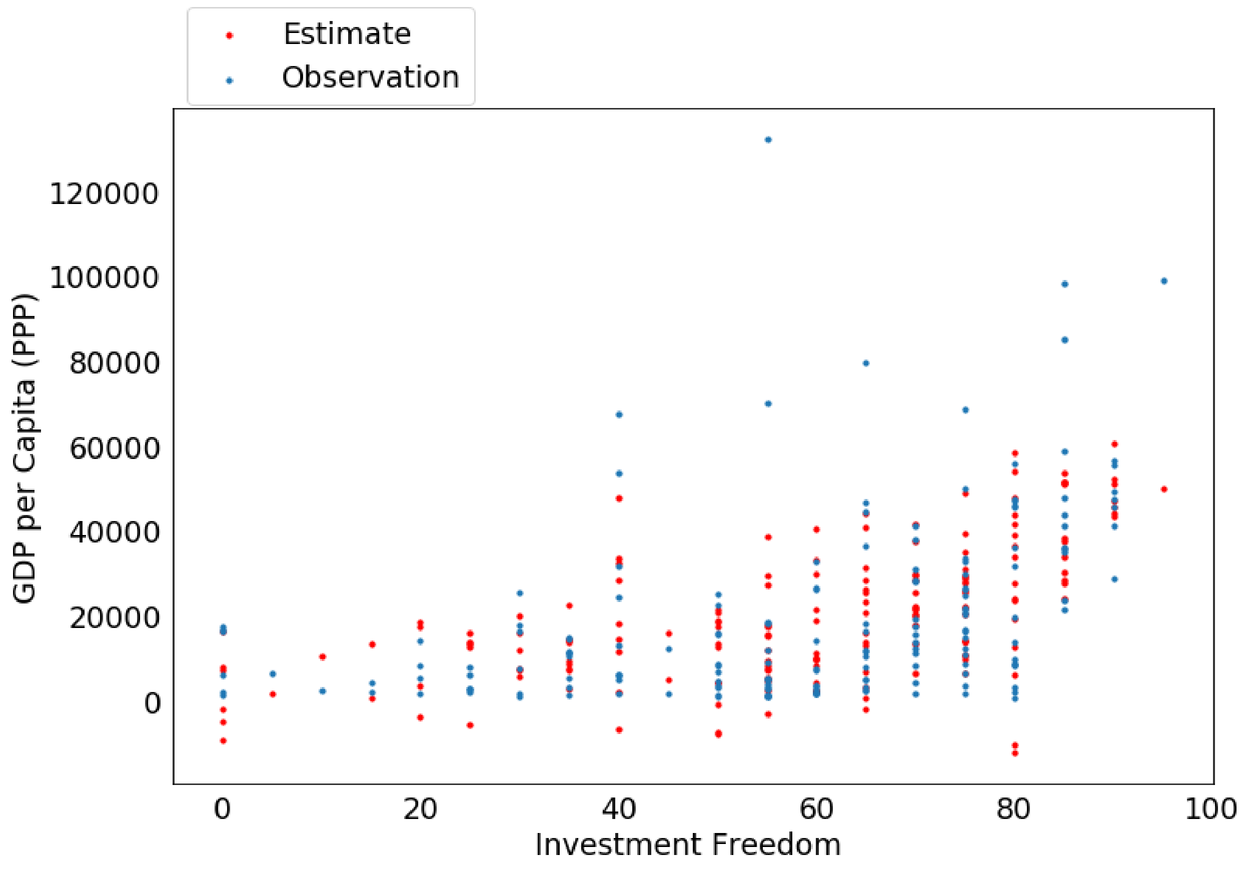
Output:

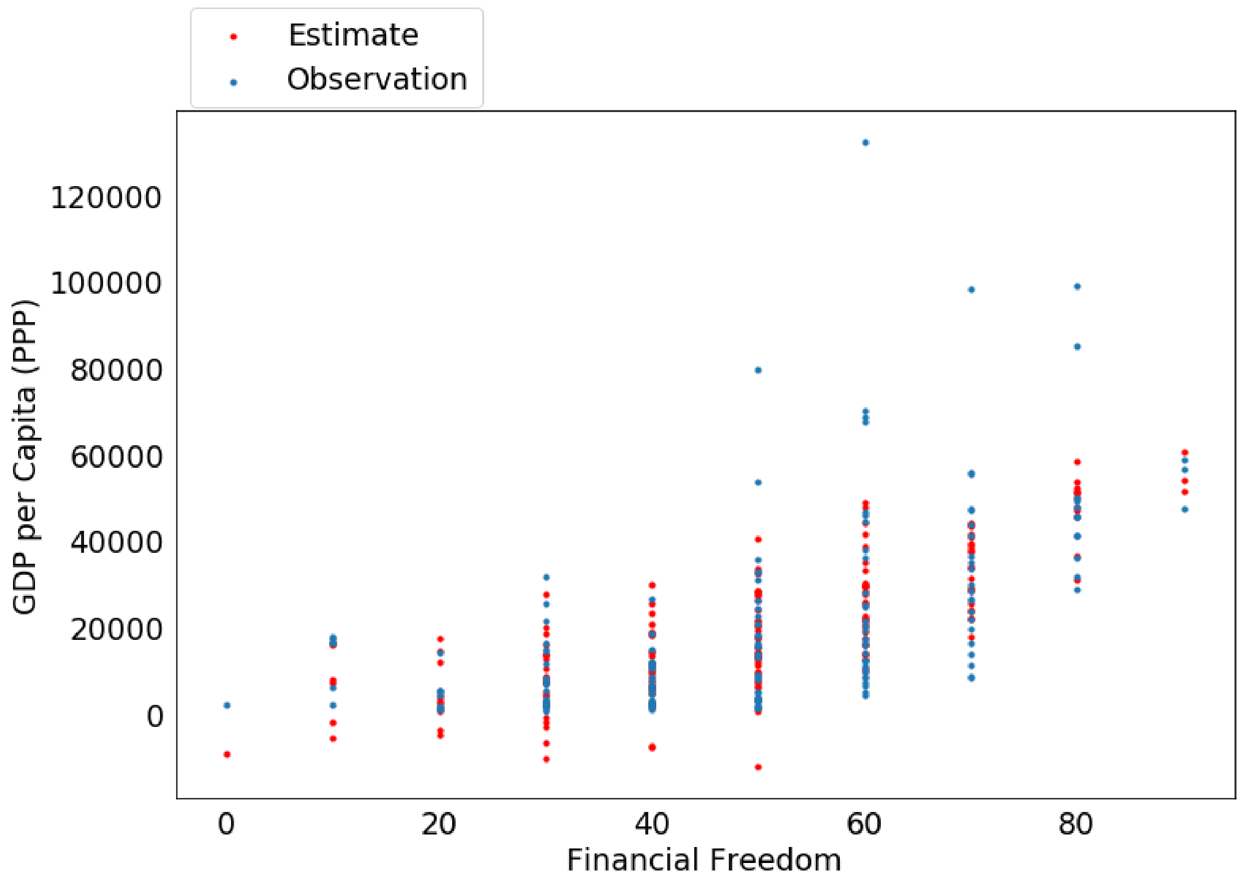



Estimate
Observation
GDP per Capita (PPP)
120000
100000
80000
60000
40000
20000
0
20
40
60
80
100
Property Rights

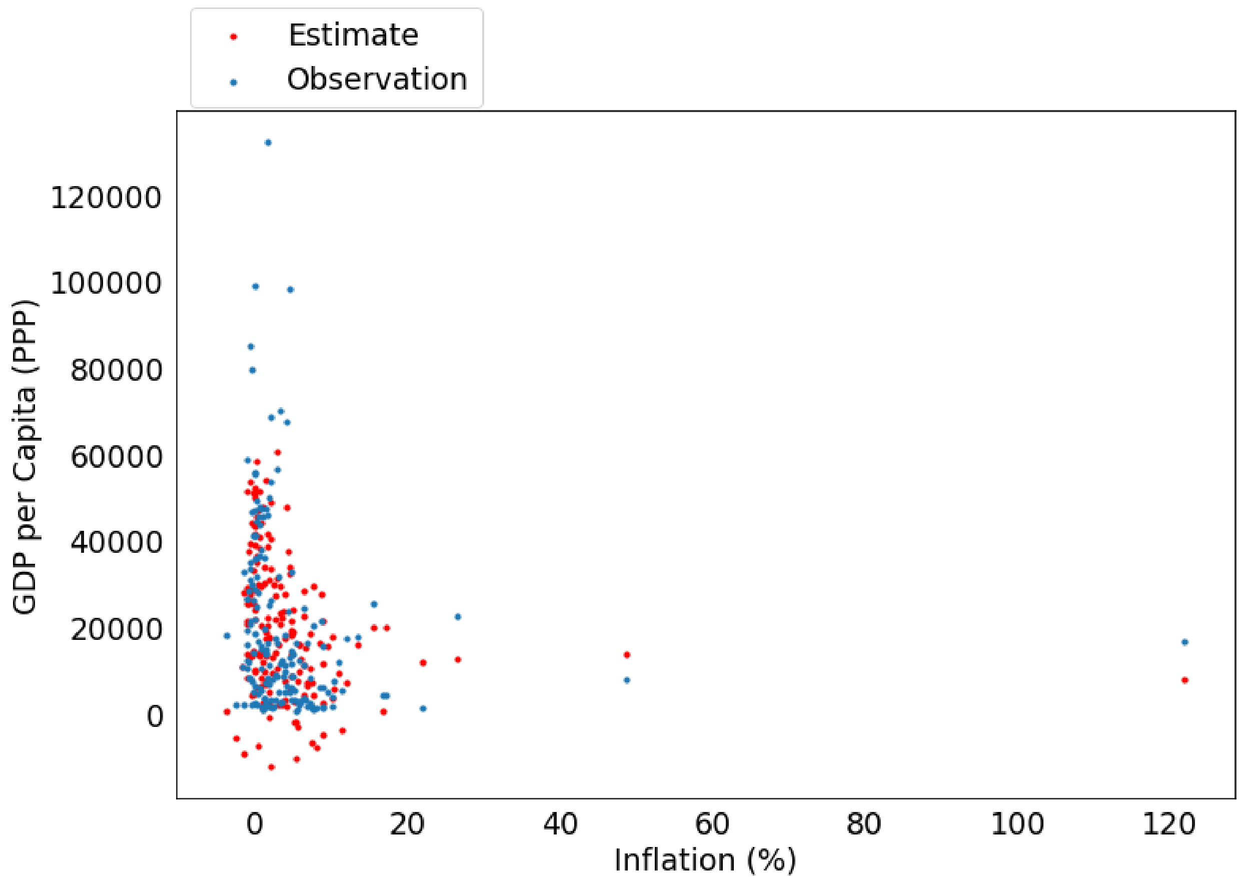


Estimate
Observation
GDP per Capita (PPP)
120000
100000
80000
60000
40000
20000
0
0
20
40
60
80
100
Fiscal Health

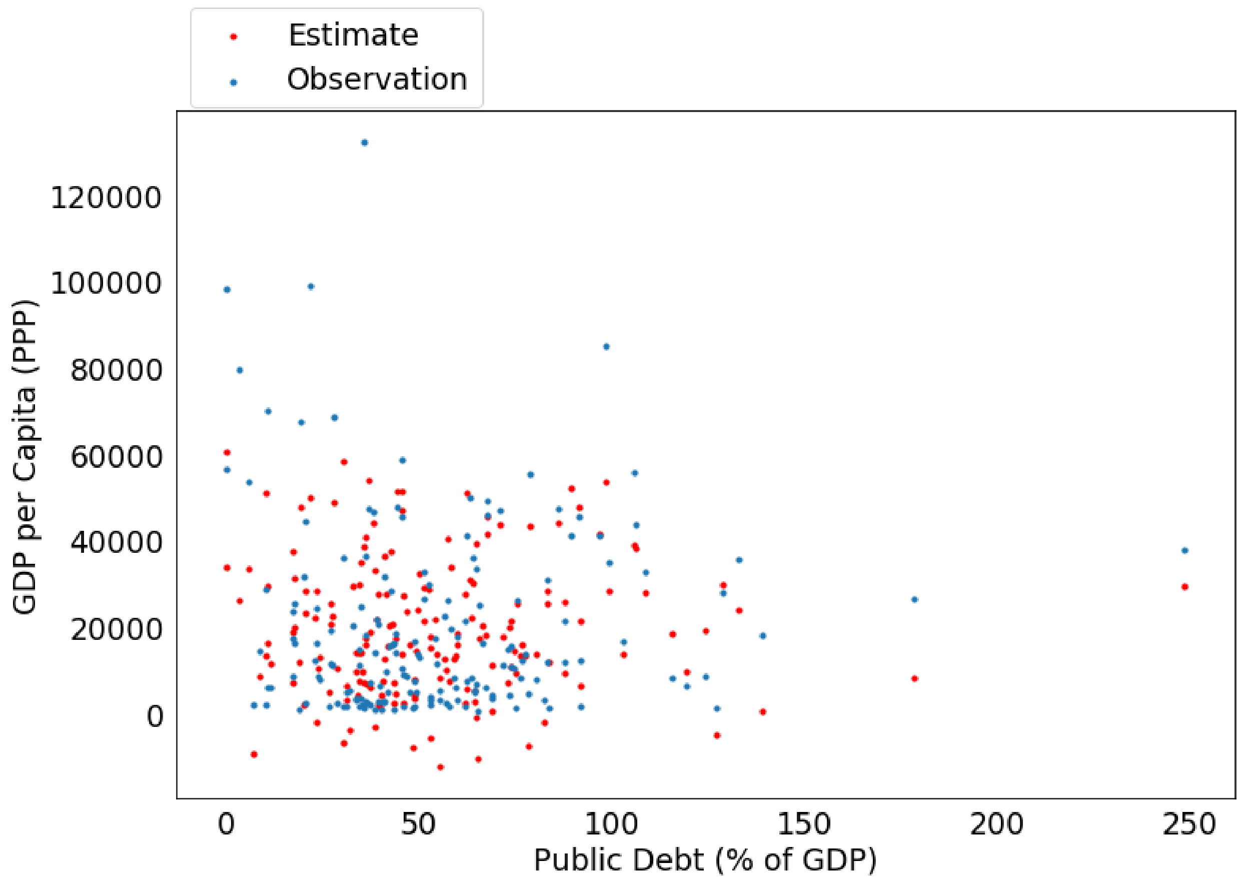

Estimate
Observation
GDP per Capita (PPP)
120000
100000
80000
60000
40000
20000
0
0
20
40
60
80
100
Investment Freedom



Estimate
Observation
GDP per Capita (PPP)
120000
100000
80000
60000
40000
20000
0
0
20
40
60
80
Financial Freedom



Estimate
Observation
GDP per Capita (PPP)
Inflation (%)



Exercise:

1. Run an OLS regression using a different set of data. Use the regression class created in this chapter. Print the results.
2. Create scatter plots of the observation and predicted values as demonstrated at the end of this chapter.
3. Use the numpy libraries log function to log some or all value in your data. Print the columns of data that have been logged. (hint: pass the appropriate list of keys to the dataframe, data[[key1,key2,key3…]])
4. Run the same regression again. Print the results. How has the significance changed?
5. Plot the new results using scatter plots as in question 2.
6. Create 2 unique visualizations of the results using matplotlib (e.g., time series predicted values and observations, plots with more than 2 variables represented such as 3D plane or changing size of dots, a plot comparing results of the logged and unlogged regression, etc…). For visualization ideas see: https://matplotlib.org/examples/.

Chapter 8: Advanced Data Analysis

In the last few chapters, we have become comfortable with the idea of building our own functions. These can become quite complex, as we have learned with the construction of the OLS regression in chapter 7. We will continue developing our tool set for working with and managing large sets of data by integrating data from different data sets. We will introduce the *multi index* to facilitate this process. Among other things, the multi index is useful for including identifiers across time and region or principality. We will also use the *multi index* to perform a panel regression that controls for level effects between different countries.

Using a Double Index to Work with Panel Data

In any project, it will not be uncommon for data to be attached to more than one indentifying category. Often, data will be labeled by polity and by date. In the next several examples, we will work with multiple data sets of this sort, working to combine different data sets, investigate the features of the double index, and use this data in a panel regression that can control for effects by polity and by time period.

Plotting with Double Index

We will be working with two datasets in the next example: the Fraser Economic Freedom Index and GDP from the Maddison Project:

Fraser Economic Freedom Index: https://www.fraserinstitute.org/economic-freedom/dataset?geozone=world&page=dataset&min-year=2&max-year=0&filter=0&year=2017

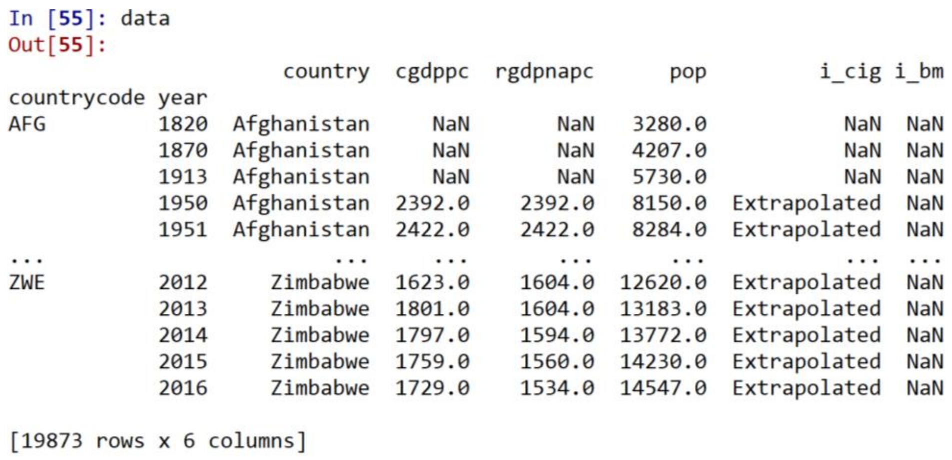
Maddison GDP: https://www.rug.nl/ggdc/historicaldevelopment/maddison/releases/maddison-project-database-2018

Due to the formatting of the Fraser Economic Freedom Index, the first column and the first three rows of data in the sheet titled “EFW Index 2018 Report” will need to be deleted. Save the edited data as *EFW2018ForPython.xlsx*. You may save the Maddison data with its original name. Both should be saved in the same folder as the script below.

First, import the GDP data from Maddison Project.

```python
#multiIndex.py
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# index_col = [0,2] will choose countrycode as primary index, and year as
# secondary index
data = pd.read_excel("mpd2018.xlsx", sheet_name = "Full data", index_col = [0,2])
```

View the new dataframe by entering *data* in the console:



```
In [55]: data
Out[55]:
                      country  cgdppc  rgdpnapc      pop        i_cig i_bm
countrycode year
AFG         1820  Afghanistan     NaN       NaN   3280.0          NaN  NaN
            1870  Afghanistan     NaN       NaN   4207.0          NaN  NaN
            1913  Afghanistan     NaN       NaN   5730.0          NaN  NaN
            1950  Afghanistan  2392.0    2392.0   8150.0 Extrapolated  NaN
            1951  Afghanistan  2422.0    2422.0   8284.0 Extrapolated  NaN
...                       ...     ...       ...      ...          ...  ...
ZWE         2012     Zimbabwe  1623.0    1604.0  12620.0 Extrapolated  NaN
            2013     Zimbabwe  1801.0    1604.0  13183.0 Extrapolated  NaN
            2014     Zimbabwe  1797.0    1594.0  13772.0 Extrapolated  NaN
            2015     Zimbabwe  1759.0    1560.0  14230.0 Extrapolated  NaN
            2016     Zimbabwe  1729.0    1534.0  14547.0 Extrapolated  NaN

[19873 rows x 6 columns]
```

When working with a multi index, calling the values from a single index requires a few steps. If you were to call *data.index* in the console, both the countrycode and the year values would be returned:

```
MultiIndex([('AFG', 1820),
      ('AFG', 1870),
      ('AFG', 1913),
      ('AFG', 1950),
      ('AFG', 1951),
      ('AFG', 1952),
      ('AFG', 1953),
      ('AFG', 1954),
      ('AFG', 1955),
      ('AFG', 1956),
      ...
      ('ZWE', 2007),
      ('ZWE', 2008),
      ('ZWE', 2009),
      ('ZWE', 2010),
      ('ZWE', 2011),
      ('ZWE', 2012),
      ('ZWE', 2013),
      ('ZWE', 2014),
      ('ZWE', 2015),
      ('ZWE', 2016)],
     names=['countrycode', 'year'], length=19873)
```

To call only the year values from the multi index, we use the dataframe method, *.get_level_values("year")*. This returns the same list, but with only years:

```
In [63]: data.index.get_level_values('year')
Out[63]:
Int64Index([1820, 1870, 1913, 1950, 1951, 1952, 1953, 1954, 1955, 1956,
            ...
            2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016],
           dtype='int64', name='year', length=19873)
```

Since we don't need to hold every repeated year value, only the full range years present in the data set. We will remove an repeated values from the list and then ensure that the list is in order:

```
9.  # create list of years to identify max extent of date range to plot
10. # First find list of all years in index
11. years = data.index.get_level_values('year')
12. # then remove any repeated observations
13. years = set(years)
14. # sort the years in numerical order
15. years = sorted(list(years))
```

The commands in lines 8-10 extract the range of years. Print the beginning and end of the list by entering the following in the console:

```
years[:10], years[-10:-1]
```

Output:

```
([1, 730, 1000, 1150, 1280, 1281, 1282, 1283, 1284, 1285],
 [2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015])
```

The data set has relatively few observations for data before the modern era. Before 1800, even for the years that have entries, data for most countries is not present.

Next, we create a dictionary with tuples containing codes for each pair of countries whose real GDP per capita we will compare. The first pair is Canada and Finland, the second pair is France and Germany, and the third pair is Great Britain and the Netherlands. The first country in each pair will be represented by the linestyle "-", and the second pair by the linestyle ":".

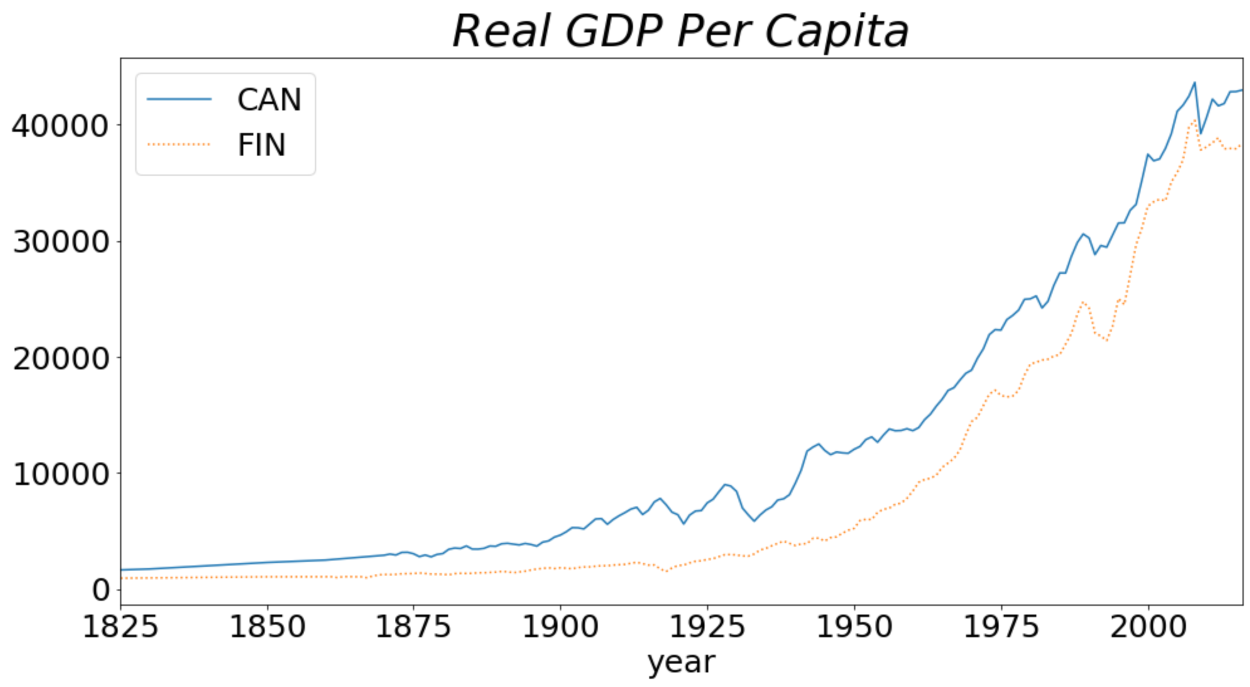
```
16.
17. #pairs of countries to compare in plots
18. pairs = [("CAN", "FIN"), ("FRA", "DEU"), ("GBR", "NLD")]
19. linestyles = ["-", ":"]
```

Now that the dictionary has been prepared, cycle through each pair of countries in the dictionary. Using a for loop, we select one of these at a time with an accompanying linestyle. The code of the country calls the Real GDP data using the command *data.ix[(country),:]["cgdppc"]*. This selects data by index according to country and includes all years, as is indicated by the colon in the second part of the index entry. Last, the column "cgdppc" is selected. Conveniently, we can also use the code stored in *country* as the label.

Script in lines 27-30 continue to adjust the plot. *The command plt.xlim(xmin, xmax)* selects the range of the x-axis. Only data from 1825 to the most recent observation, identified by max(years), is included in the plot. The *plt.rcParams.update()* commands adjust fontsizes by accessing default parameter in matplotlib. Finally, the create a title that uses latex text by including "*$text$*". This itlacizes the title text and allows for the use latex commands such as subscripting, greek letters, etc....

```

for pair in pairs:
    fig, ax = plt.subplots(figsize=(16,8))
    for i in range(len(pair)):
        country = pair[i]
        linestyle = linestyles[i]
        data.ix[country,:]["cgdppc"].dropna().plot.line(ax = ax,
                 label = country, linestyle = linestyle)
    plt.xlim([1825, max(years)])
    plt.rcParams.update({"legend.fontsize": 25, "legend.handlelength": 2})
    plt.rcParams.update({"font.size": 25})
    plt.title("$Real$ $GDP$ $Per$ $Capita$", fontsize=36)
    plt.legend()
    plt.show()
    plt.close()
```

The result should include plots like the following:



Merge Data Sets with Double Index

Next, we will import both sets of data as unique objects with the intention of combining them. First, we must successful import the Fraser Index. Unlike with other files we have imported, the column names are in the third row and the years are in the second column. Finally, we will call the *.dropna()* method twice to drop any row (*axis = 0*) and then any column (*axis=1*) with missing observations:

```
#dataForPanel.py
```

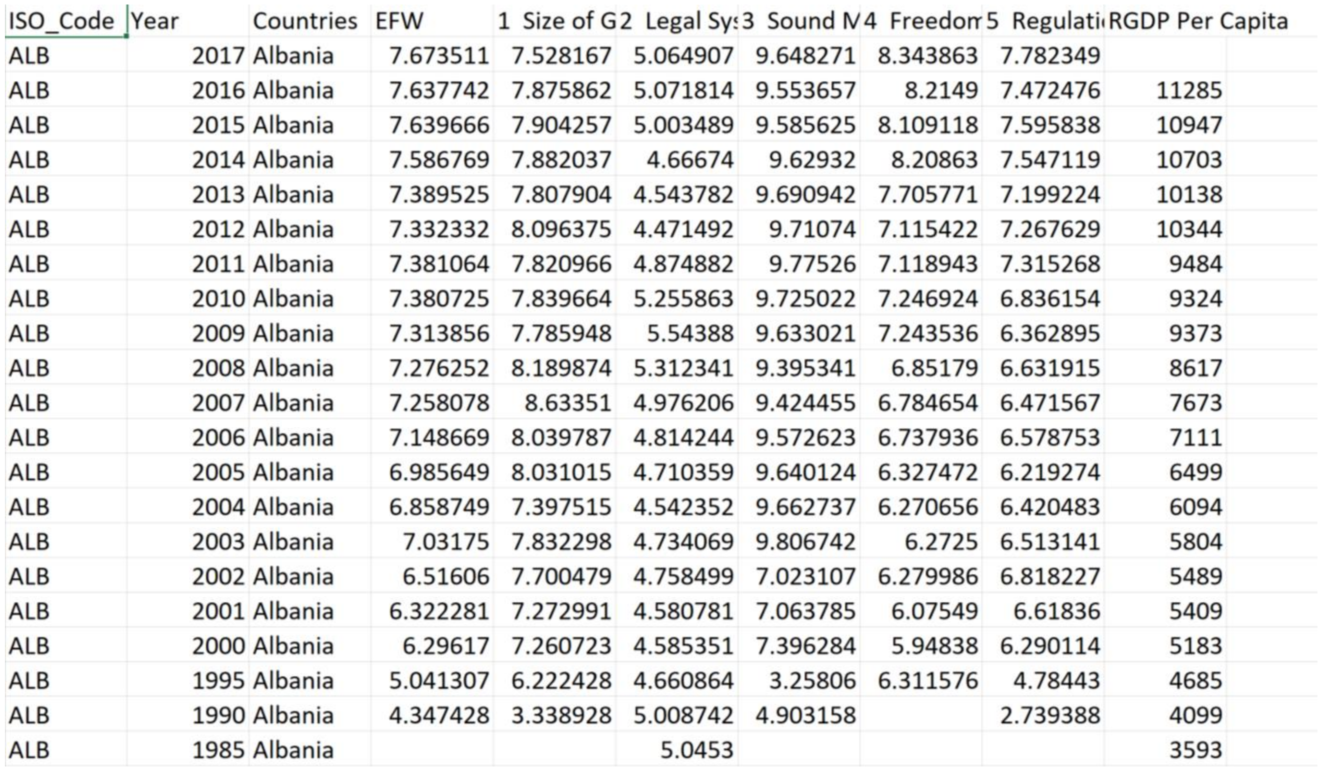
```
import pandas as pd

#make sure dates are imported in the same format; to do this,
#we turned off parse_dates
fraser_data = pd.read_excel("efw-2019-master-index-data-for-researchers.xlsx",
                           sheet_name = "EFW Panel Data 2019 Report",
                           header = [2], index_col = [2, 1], parse_dates=False)
# drop any empty column and any empty row
fraser_data = fraser_data.dropna(axis=0, thresh=1).dropna(axis=1, thresh=1)

maddison_data = pd.read_excel("mpd2018.xlsx", sheet_name = "Full data",
                              index_col = [0,2])
```

The object *fraserData* is imported from the sheet “EFW Index 2018 Report”. The 1st and 0th columns are used as index columns in that order, respectively. The object *maddisonData* is imported from the sheet “Full data”. The 0th and 2nd columns columns are used as index columns. The double index of both dataframes refers to the ISO_Code (countrycode) and the year. Since the objects referred to by the indices match, we will be able to use the double index associated with a particular entry to refer to another double index.

Since both dataframes employ the same double index format, we can copy any column from one dataframe to the other. We will copy for the Maddison GDP data to the Fraser Economic Freedom Index data.

```

fraser_data["RGDP Per Capita"] = maddison_data["cgdppc"]
fraser_data.to_csv("fraserDataWithRGDPPC.csv")
```

Now that the dataset has been transferred, we can save the result as a csv so that we can call it again in later examples. RGDP Per Capita is shown in the last column of the new csv file:



ISO_Code	Year	Countries	EFW	1 Size of G	2 Legal Sys	3 Sound M	4 Freedom	5 Regulati	RGDP Per Capita
ALB	2017	Albania	7.673511	7.528167	5.064907	9.648271	8.343863	7.782349	
ALB	2016	Albania	7.637742	7.875862	5.071814	9.553657	8.2149	7.472476	11285
ALB	2015	Albania	7.639666	7.904257	5.003489	9.585625	8.109118	7.595838	10947
ALB	2014	Albania	7.586769	7.882037	4.66674	9.62932	8.20863	7.547119	10703
ALB	2013	Albania	7.389525	7.807904	4.543782	9.690942	7.705771	7.199224	10138
ALB	2012	Albania	7.332332	8.096375	4.471492	9.71074	7.115422	7.267629	10344
ALB	2011	Albania	7.381064	7.820966	4.874882	9.77526	7.118943	7.315268	9484
ALB	2010	Albania	7.380725	7.839664	5.255863	9.725022	7.246924	6.836154	9324
ALB	2009	Albania	7.313856	7.785948	5.54388	9.633021	7.243536	6.362895	9373
ALB	2008	Albania	7.276252	8.189874	5.312341	9.395341	6.85179	6.631915	8617
ALB	2007	Albania	7.258078	8.63351	4.976206	9.424455	6.784654	6.471567	7673
ALB	2006	Albania	7.148669	8.039787	4.814244	9.572623	6.737936	6.578753	7111
ALB	2005	Albania	6.985649	8.031015	4.710359	9.640124	6.327472	6.219274	6499
ALB	2004	Albania	6.858749	7.397515	4.542352	9.662737	6.270656	6.420483	6094
ALB	2003	Albania	7.03175	7.832298	4.734069	9.806742	6.2725	6.513141	5804
ALB	2002	Albania	6.51606	7.700479	4.758499	7.023107	6.279986	6.818227	5489
ALB	2001	Albania	6.322281	7.272991	4.580781	7.063785	6.07549	6.61836	5409
ALB	2000	Albania	6.29617	7.260723	4.585351	7.396284	5.94838	6.290114	5183
ALB	1995	Albania	5.041307	6.222428	4.660864	3.25806	6.311576	4.78443	4685
ALB	1990	Albania	4.347428	3.338928	5.008742	4.903158		2.739388	4099
ALB	1985	Albania			5.0453				3593

Creating Indicator Variables

Suppose that we wanted to test the idea that geography influences economic growth. We would need to clarify a hypothesis concerning this. We might believe, for example, that countries in North America tend to have a distinct real gross domestic product than in other continents i.e., real GDP tends to be higher or lower due to residing in North America. To represent this, we would create an indicator variable named "North America". Countries residing in North America would be indicated with a 1 (i.e., True), and those outside of North America would receive a zero.

To accomplish this task is straightforward if you know the appropriate commands to use. As usual, we import the data. Before creating an indicator variable, you will need to choose the name that will reference the indicator variable, *indicator_name*, and make a list of the index values, *target_index_list* that will be recorded as possessing the attribute referred to by the indicator variable. Finally, you will need to choose the name of the index column that includes the elements in the *target_index_list*. If you are not sure what this name is, you can check the names of the index columns using:

```
data.index.names
```

Output:

FrozenList(['ISO_Code', 'Year'])

For this purpose, we are interested in identifying countries, not years, so “ISO_Code” is the appropriate index column.

```python
#indicatorVariable.py
import pandas as pd

def create_indicator_variable(data, indicator_name, index_name,
                              target_index_list):
    # Prepare column with name of indicator variable
    data[indicator_name] = 0
    # for each index whose name matches an entry in target_index_list
    # a value of 1 will be recorded
    for index in target_index_list:
        data[indicator_name].loc[data.index.get_level_values(\
            index_name) == index] = 1

# Import data with "ISO_Code" and "Year" as index columns
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)

# select "ISO_Code" from names of double index
index_name = data.index.names[0]
indicator_name = "North America"
# Cuba, Grenada, Saint Kitts, Saint Lucia, Saint Vincent are missing
# from Fraser Data
countries_in_north_america = ["BHS", "BRB", "BLZ", "CAN", "CRI", "DOM", "SLV",
                              "GTM", "HTI", "HND", "JAM", "MEX", "NIC", "PAN",
                              "TTO", "USA"]
create_indicator_variable(data = data, indicator_name = indicator_name,
    index_name = index_name, target_index_list = countries_in_north_america)
```

Once the new column of data has been creative, it can be called by passing *indicator_name* to the dataframe. Be sure to use double brackets (i.e., [[,]]) so that the column names are included in the output.

```
In [114]: data[["North America"]]
Out[114]:
                       North America
ISO_Code Year
ALB      2017-01-01                0
         2016-01-01                0
         2015-01-01                0
         2014-01-01                0
         2013-01-01                0
...                              ...
ZWE      1990-01-01                0
         1985-01-01                0
         1980-01-01                0
         1975-01-01                0
         1970-01-01                0

[3888 rows x 1 columns]
```

Create Quantile Ranking

Similar to the pervious exercise, we may categorize data according ranked bins. This is accomplished by separating data into quantiles, often in the form of quartiles or quintiles, however this can be accomplished using any number of divisions. The following script allows you to create quantiles of the number of divisions of your choosing.

First, we need to import the data that we have merged. Once the dataframe is created, we must prepare a place for quantile data to be registered. Out of convenience, we will refer to the n-tile, meaning that a quartile will be labeled a “4-tile”, a quintile will be label a “5-tile”, and so forth. Before recording the data, we will create blank entries using *np.nan*.

```python
#quantile.py
import pandas as pd
import numpy as np

# choose numbers of divisions
n = 5
# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
#create column identifying n-tile rank
quantile_var = "RGDP Per Capita"
quantile_name = quantile_var + " " + str(n) + "-tile"
data[quantile_name] = np.nan
```

Now that the dataframe has been prepped, we can create function that will record quantile values. Since the data frame includes data for nearly every country over many years, we will want to choose one of the index categories to identify quantiles. We prefer to compare countries, 8so unique quantile measures will compare countries in a given year. We will build a list of years and use a for-loop to cycle through values in the list. For each year, we will construct a quantile values comparing countries within the year.

The *create_quantile()* function needs to be provided the number of divisions within a quantile (*n*), the dataframe (*data*), the year to which the quantile measure refers (*year*), the variable for which the quantile is constructed (*quantile_var*), and the key that will refer to the quantile data (*quantile_name*).

First, we construct the *year_index*. This provides a slice of the original index that includes only the years of interest. Next, we identify the value that divides each quantile. The *pandas* dataframe has a convenient command, *df.quantile(percent)*, that will calculate the value for a particular percentile. For example, *df.quantile(.25)* will calculate the value representing the 25th percentile. A quantile is comprised of divisions whose size is a fraction of 1 and that sum to one. The value that divides each quantile is defined by $\frac{i}{n}$ where n is the number of quartiles, and *i* includes all integers from *1* up to and including *n*.

Now that the values dividing each quantile for a given year have been identified, we can check which in which quantile each country falls. Cycle through the index for the year with *for index in* *data[year_index].index*. This allows us to calls up each individual countries data for the given year, compare that data of the category of interest to the quantile values, and select the identify quantile that the nation falls in for the year.

```python
#quantile.py
import pandas as pd
import numpy as np

def create_quantile(n, data, year, quantile_var, quantile_name):
    # index that indentifies countries for a given year
    year_index = data.index.get_level_values("Year") == year
    quantile_values_dict = {i:data[year_index][quantile_var]
                            .quantile(i/n) for i in range(1, n + 1)}
    # cycle through each country for a given year
    for index in data[year_index].index:
        # identtify value of the variable of interest
        val = data.ix[index][quantile_var]
        # compare that value to the values that divide each quantile
        for i in range(1, n + 1):
            # if the value is less than the highest in the quantile identified,
            # save quantile as i
            if val <= quantile_values_dict[i]:
                data[quantile_name][index]=int(i)
                #exit loop
                break
            # otherwise check the higest value of the next quantile
        else:
            continue

# choose numbers of divisions
n = 5
# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
#create column identifying n-tile rank
quantile_var = "RGDP Per Capita"
quantile_name = quantile_var + " " + str(n) + "-tile"
data[quantile_name] = np.nan
years = list(sorted(set(data.index.get_level_values("Year"))))
for year in years:
    create_quantile(n, data, year, quantile_var, quantile_name)
```

We can call the new column with the *"Real GDP Per Capita"*:

```
In [122]: data[["RGDP Per Capita", "RGDP Per Capita 5-tile"]]
Out[122]:
                     RGDP Per Capita  RGDP Per Capita 5-tile
ISO_Code Year
ALB      2017-01-01              NaN                     NaN
         2016-01-01          11285.0                     3.0
         2015-01-01          10947.0                     3.0
         2014-01-01          10703.0                     3.0
         2013-01-01          10138.0                     3.0
...                              ...                     ...
ZWE      1990-01-01           3265.0                     2.0
         1985-01-01           3615.0                     2.0
         1980-01-01           4003.0                     3.0
         1975-01-01           4142.0                     3.0
         1970-01-01           3448.0                     3.0

[3888 rows x 2 columns]
```

Lag Variables and Differenced Log Values

With time series data, it is often useful to control for trends when data is autocorrelated. Consider, for example, that real GDP data is often highly correlated with values from the previous period. We might detect a false causal relationship between two variables that are actually unrelated but follow a similar trend. For example, we might regress your age against real GDP and find that there is a strong correlation between the two. To avaoid false positives like this, it is useful to account for the influence of lagged values and/or to detrend the data all together by using differenced logs.

Creating lag variables is quite simple if the index is already recognized as containing dates and times. The method, *df.shift(n)* accomplishes this. Pass a negative value to create a lagged variable from n periods previous and a positive value to create a variable that refers to data *n* periods in the future.

Because we are using a double index, we must specify to which index we must instruct Pandas as to which index the shift refers. We accomplish this by using *.groupby(level)* to target the index column that does *not* refer to a datetime data. This will group the data by entity, thus leaving only the date column to be referenced by *.shift(n)*. Since we only want a lag value from the period that immediately preceded the observation, *n=-1*.

```python
#logAndDifferenceData.py
import pandas as pd
import numpy as np
import datetime

# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level="ISO_Code")\
    ["RGDP Per Capita"].shift(-1)
```

To understand what the command yielded, we will want to view data for an individual country. The following command will save "RGDP Per Capita" in the United States and the lag of that value to a csv file:

```
In [142]: data.loc["USA", :]
Out[142]:
                 Countries       EFW  ...  RGDP Per Capita  RGDP Per Capita Lag
Year                                  ...
2017-01-01  United States  8.196365  ...              NaN              53015.0
2016-01-01  United States  8.183442  ...          53015.0              52591.0
2015-01-01  United States  8.085664  ...          52591.0              51664.0
2014-01-01  United States  7.969703  ...          51664.0              50863.0
2013-01-01  United States  7.900998  ...          50863.0              50394.0
2012-01-01  United States  8.012288  ...          50394.0              49675.0
2011-01-01  United States  7.906265  ...          49675.0              49267.0
2010-01-01  United States  7.972577  ...          49267.0              48453.0
2009-01-01  United States  7.965853  ...          48453.0              50276.0
2008-01-01  United States  8.244885  ...          50276.0              50902.0
2007-01-01  United States  8.394524  ...          50902.0              50490.0
2006-01-01  United States  8.318497  ...          50490.0              49655.0
2005-01-01  United States  8.353783  ...          49655.0              48493.0
2004-01-01  United States  8.443225  ...          48493.0              47158.0
2003-01-01  United States  8.454976  ...          47158.0              46267.0
2002-01-01  United States  8.457409  ...          46267.0              45878.0
2001-01-01  United States  8.410677  ...          45878.0              45887.0
2000-01-01  United States  8.518073  ...          45887.0              39391.0
1995-01-01  United States  8.341947  ...          39391.0              36982.0
1990-01-01  United States  8.243760  ...          36982.0              33024.0
1985-01-01  United States  7.953914  ...          33024.0              29613.0
1980-01-01  United States  7.770014  ...          29613.0              25956.0
1975-01-01  United States  7.586137  ...          25956.0              23958.0
1970-01-01  United States  7.450927  ...          23958.0                  NaN

[24 rows x 9 columns]
```

Since not every year is included in the index, this actually results in false values for periods where observations are only available once every 5 years. If we use lagged values, we need to delimit the data to consecutive annual observations. Since consistent data is provided starting in the year 2000. To select data by year, we need to inicate that we wish to form selection criteria that refers to values form the year column in the index. The command *data.index.get_level_Values("Year")* calls these values. Once we execute the above script, we can call this command in the console.

```
data.index.get_level_Values("Year")
```

Output:

```
DatetimeIndex(['2016-01-01', '2016-01-01', '2016-01-01', '2016-01-01',
               '2016-01-01', '2016-01-01', '2016-01-01', '2016-01-01',
               '2016-01-01', '2016-01-01',
               ...
               '1970-01-01', '1970-01-01', '1970-01-01', '1970-01-01',
               '1970-01-01', '1970-01-01', '1970-01-01', '1970-01-01',
               '1970-01-01', '1970-01-01'],
              dtype='datetime64[ns]', name='Year', length=3726, freq=None)
```

If we compare these values to the critierion specified, a column of boolean values will be generated that identifies where values from the index meet the specified criterion.

```
data.index.get_level_Values("Year") > datetime.datetime(2000, 1, 1)
```

Output:

```
        array([ True,  True,  True, ..., False, False, False])
```

If we pass the command, data.index.get_level_values("Year") > datetime.datetime(1999,1,1), to the dataframe, only observations that meet the criterion indicated will be included. In this case, any observation generated in years after 1999 will be included.

```
#logAndDifferenceData.py
import pandas as pd
import numpy as np
import datetime

# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level="ISO_Code")\
    ["RGDP Per Capita"].shift(-1)
data = data[data.index.get_level_values("Year") > datetime.datetime(2000,1,1)]
```

Call data in the console to see that the range of dates included has narrowed, thereby removing the observations separated by 5 year intervals:

```
In [148]: data
Out[148]:
                    Countries       EFW  ...  RGDP Per Capita  RGDP Per Capita Lag
ISO_Code Year                            ...
ALB      2017-01-01   Albania  7.673511  ...              NaN              11285.0
         2016-01-01   Albania  7.637742  ...          11285.0              10947.0
         2015-01-01   Albania  7.639666  ...          10947.0              10703.0
         2014-01-01   Albania  7.586769  ...          10703.0              10138.0
         2013-01-01   Albania  7.389525  ...          10138.0              10344.0
...                       ...       ...  ...              ...                  ...
ZWE      2005-01-01  Zimbabwe  2.889240  ...           1660.0               1813.0
         2004-01-01  Zimbabwe  3.181057  ...           1813.0               1958.0
         2003-01-01  Zimbabwe  3.675260  ...           1958.0               2376.0
         2002-01-01  Zimbabwe  3.625483  ...           2376.0               2624.0
         2001-01-01  Zimbabwe  3.633489  ...           2624.0               2696.0

[2754 rows x 9 columns]
```

The earliest dates included nowonly reach as far back as the year 2001.

Now that we've delimited the data, let's perform some other transformation that will help to prepare the data for a regression. It will be useful to log GDP values as the rate of growth of GDP measured in raw units tends to increase over time. By logging data, beta estimates will approximate the cross elasticity of the endogenous variable *y* with respect to a change in an exogenous variable *x*.

Since we have already created a lagged measure of real GDP, this is a good opportunitiy to use a for loop. We only want to log values referred to be a key that includes the string "GDP". We check each key for inclusion of this term. If the term is included in the key, than we add a logged version of the variable.

```python
#logAndDifferenceData.py
import pandas as pd
import numpy as np
import datetime

# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level="ISO_Code")\
    ["RGDP Per Capita"].shift(-1)
data = data[data.index.get_level_values("Year") > datetime.datetime(2000,1,1)]

for key in data:
    if "GDP" in key:
        data["Log " + key] = np.log(data[key])
```

The data is almost prepared. We have only left to create data that has been diferenced. Differenced log values approximate the rate of change of a variable. Likewise, we can difference the index values to test for the effect of an improvement in a nation's economic freedom score on the rate of GDP growth.

To difference data with a multi index, we must first organize the data so that observations are presented sequentially by entity. This uses the same command, *.groupby()*, that we used earlier to create a lag value. The result this time is that a new dataframe is created by taking the differenced values of all variables in the dataframe.

We save the new results in a dictionary that holds both the original dataframe and the new dataframe with differenced data.

```python
#logAndDifferenceData.py
import pandas as pd
import numpy as np
import datetime

# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level="ISO_Code")\
    ["RGDP Per Capita"].shift(-1)
data = data[data.index.get_level_values("Year") > datetime.datetime(2000,1,1)]

for key in data:
    if "GDP" in key:
        data["Log " + key] = np.log(data[key])
# We do not want to difference the index values, only the Real GDP values
# so initialize the diff data as teh dataframe but only include index values
# from a differenced matrix (one year of observations will be missing)
diff_index = data.groupby(level=0).diff(-1).dropna().index
data_dict = {}
data_dict["Data"] = data
data_dict["Diff Data"] = data.copy().loc[diff_index]
for key in data:
    if "GDP" in key:
        data_dict["Diff Data"][key] = data[key].groupby(level=0).diff(-1)
data_dict["Diff Data"] = data_dict["Diff Data"].dropna()
```

```
In [152]: data_dict["Diff Data"]
Out[152]:
                     Countries  ...  Log RGDP Per Capita Lag
ISO_Code Year                   ...
ALB      2016-01-01    Albania  ...                 0.022541
         2015-01-01    Albania  ...                 0.054233
         2014-01-01    Albania  ...                -0.020116
         2013-01-01    Albania  ...                 0.086800
         2012-01-01    Albania  ...                 0.017014
...                        ...  ...                      ...
ZWE      2006-01-01   Zimbabwe  ...                -0.088165
         2005-01-01   Zimbabwe  ...                -0.076941
         2004-01-01   Zimbabwe  ...                -0.193495
         2003-01-01   Zimbabwe  ...                -0.099281
         2002-01-01   Zimbabwe  ...                -0.027069

[2031 rows x 11 columns]
```

Notice that there exist no data for the first year of observations. Without a previous year to draw from for the year 1999, the *pandas* dataframe is left blank for differenced values in this year.

Using Indicator Variables in Regression

Using the function created above, we can prepare indicator and quantile variables to be used in a regression. As before, the a regression follows the same form as a standard multiple regression with continuous exogenous variables, but also include a parameter, α_k, for each indicator or quantile variables.

$$y_i = \beta_0 + \sum_{j=1}^{n} \beta_j x_{i,j} + \sum_{k=1}^{m} \alpha_k x_{i,k} + e_i$$

For both cases, we can use the regression method that we created last class. We begin with use of indicator variables in regression. An indicator variable accounts for a level effect that is attributed to a predicted value due to an attribute of the observation. Suppose that we wanted to measure whether or not there is an effect of gender on wages in a particular field once all other relevant factors are accounted for. Observations where the wage earner is a woman would be indicated with a one. The beta value estimated by a regression would indicate the effect of gender on wage given the control variables included in the regression.

We can perform a similar regression by identifying the effect of being a nation in North America on real GDP per capita of countries residing on the continent. In reality, this is actually a poor indicator variable, but we can expand our computational toolbelt by creating an indicator variable that identifies countries in North America and by including this variable in a regression. We will see that the creation of indicator variables is fundamental to the panel regression.

```python
#indicatorRegression.py
import pandas as pd
import numpy as np
import datetime

def create_indicator_variable(data, indicator_name, index_name,
                              target_index_list):
    # Prepare column with name of indicator variable
    data[indicator_name] = 0
    # for each index whose name matches an entry in target_index_list
    # a value of 1 will be recorded
    for index in target_index_list:
        data[indicator_name].loc[data.index.get_level_values(\
            index_name) == index] = 1

# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level="ISO_Code")\
    ["RGDP Per Capita"].shift(-1)
data = data[data.index.get_level_values("Year") > datetime.datetime(2000,1,1)]

for key in data:
    if "GDP" in key:
        data["Log " + key] = np.log(data[key])
# We do not want to difference the index values, only the Real GDP values
# so initialize the diff data as teh dataframe but only include index values
# from a differenced matrix (one year of observations will be missing)
diff_index = data.groupby(level=0).diff(-1).dropna().index
data_dict = {}
data_dict["Data"] = data
data_dict["Diff Data"] = data.copy().loc[diff_index]
for key in data:
    if "GDP" in key:
        data_dict["Diff Data"][key] = data[key].groupby(level=0).diff(-1)
data_dict["Diff Data"] = data_dict["Diff Data"].dropna()

# Create indicator variable for North Amierca in both data and diff_data
indicator_name = "North America"
index_name = "ISO_Code"
countries_in_north_america = ["BHS", "BRB", "BLZ", "CAN", "CRI", "DOM", "SLV",
                              "GTM", "HTI", "HND", "JAM", "MEX", "NIC", "PAN",
                              "TTO", "USA"]
for key in data_dict:
    data = data_dict[key]
    create_indicator_variable(data = data, indicator_name = indicator_name,
        index_name = index_name,target_index_list = countries_in_north_america)
```

As in the earlier example, we have created indicator variables. Notice that the created are not differenced even in the dataframe with differenced data. We will be using these values in regressions where the estimated value is either logged or log-differenced. In the first case, the indicator variable will effect the level estimated. In the second case, the indicator variable influences an estimated rate.

We've left to estimate regression using data from each dataframe. We will estimate the impact of economic freedom ("SUMMARY INDEX") on Real GDP Per Capita. To control for autocorrelation, we include the lag of Real GDP Per Capita. To run the regression once for each dataframe in *data_dict*, we use a for loop that cycles through the keys in *data_dict*. We print the statistics for the estimated beta values as well as statistics that describe the results of the regression overall. The *key* is also printed to identify each set of results.

```
#indicatorRegression.py
import pandas as pd
import numpy as np
import datetime
import regression

def create_indicator_variable(data, indicator_name, index_name,
                              target_index_list):
    # Prepare column with name of indicator variable
    data[indicator_name] = 0
    # for each index whose name matches an entry in target_index_list
    # a value of 1 will be recorded
    for index in target_index_list:
        data[indicator_name].loc[data.index.get_level_values(\
            index_name) == index] = 1

# import data
data = pd.read_csv("fraserDataWithRGDPPC.csv", index_col = ["ISO_Code", "Year"],
                   parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level="ISO_Code")\
    ["RGDP Per Capita"].shift(-1)
data = data[data.index.get_level_values("Year") > datetime.datetime(2000,1,1)]

for key in data:
    if "GDP" in key:
        data["Log " + key] = np.log(data[key])
# We do not want to difference the index values, only the Real GDP values
# so initialize the diff data as teh dataframe but only include index values
# from a differenced matrix (one year of observations will be missing)
diff_index = data.groupby(level=0).diff(-1).dropna().index
data_dict = {}
data_dict["Data"] = data
data_dict["Diff Data"] = data.copy().loc[diff_index]
for key in data:
    if "GDP" in key:
        data_dict["Diff Data"][key] = data[key].groupby(level=0).diff(-1)
data_dict["Diff Data"] = data_dict["Diff Data"].dropna()

# Create indicator variable for North Amierca in both data and diff_data
indicator_name = "North America"
index_name = "ISO_Code"
countries_in_north_america = ["BHS", "BRB", "BLZ", "CAN", "CRI", "DOM", "SLV",
                              "GTM", "HTI", "HND", "JAM", "MEX", "NIC", "PAN",
44.                               "TTO", "USA"]
45. for key in data_dict:
46.     data = data_dict[key]
47.     create_indicator_variable(data = data, indicator_name = indicator_name,
48.         index_name = index_name,target_index_list = countries_in_north_america)
49.
50. # prepare regression variables
51. X_names = ["EFW", "Log RGDP Per Capita Lag"]
52. y_name = ["Log RGDP Per Capita"]
53.
54. # save instance of regression class
55. reg = regression.Regression()
56. for key in data_dict:
57.     # call OLS method
58.     data = data_dict[key]
59.     reg.OLS(reg_name = key, data = data.dropna(),
60.             y_name = y_name, beta_names = X_names)
61.     print(key, reg.estimates, sep = "\n")
62.     print(reg.stats_DF)
63.     print()
```

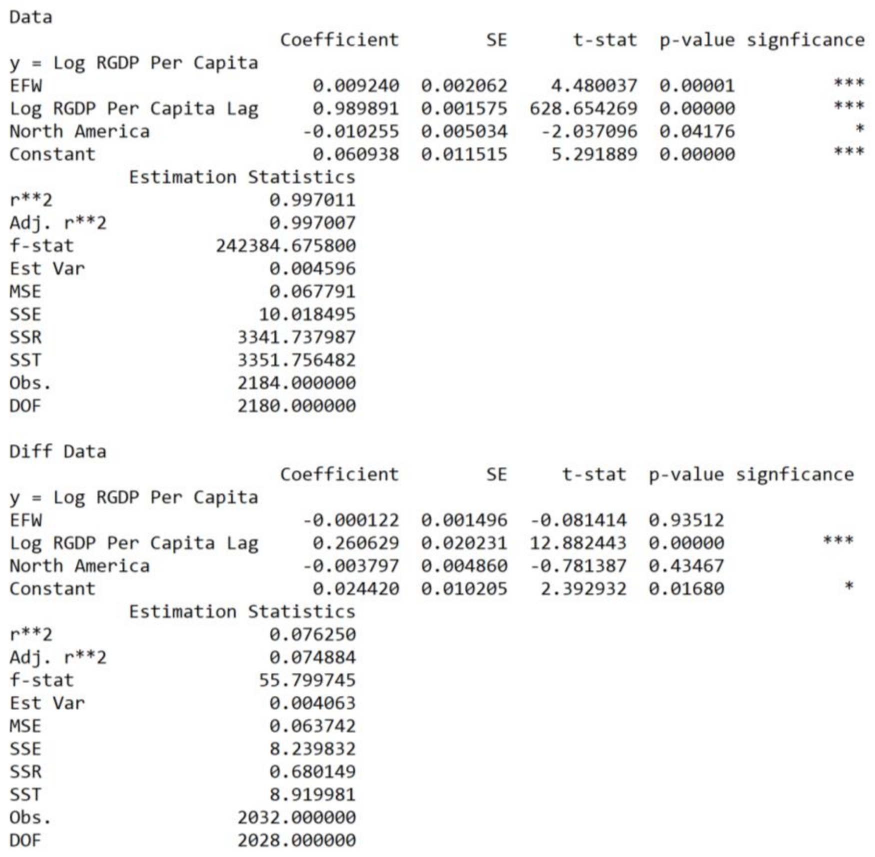
Output:

```
Data
                         Coefficient        SE       t-stat  p-value signficance
y = Log RGDP Per Capita
EFW                         0.008443  0.002026    4.166295  0.00003         ***
Log RGDP Per Capita Lag     0.990251  0.001566  632.417010  0.00000         ***
Constant                    0.062085  0.011510    5.394053  0.00000         ***
           Estimation Statistics
r**2                    0.997005
Adj. r**2               0.997003
f-stat             363050.627707
Est Var                 0.004602
MSE                     0.067840
SSE                    10.037566
SSR                  3341.718917
SST                  3351.756482
Obs.                 2184.000000
DOF                  2181.000000

Diff Data
                         Coefficient        SE      t-stat  p-value signficance
y = Log RGDP Per Capita
EFW                        -0.000299  0.001479   -0.202517  0.83953
Log RGDP Per Capita Lag     0.261129  0.020219   12.914891  0.00000         ***
Constant                    0.025245  0.010149    2.487290  0.01295           *
           Estimation Statistics
r**2                    0.075972
Adj. r**2               0.075061
f-stat                 83.410345
Est Var                 0.004062
MSE                     0.063736
SSE                     8.242313
SSR                     0.677668
SST                     8.919981
Obs.                 2032.000000
DOF                  2029.000000
```

We can check if the indicator variable, “North America”, adds any explanatory value by adding the variable to this regression. Only line 43 is changed by this addition.

```
50. # prepare regression variables
51. X_names = ["SUMMARY INDEX", "Log RGDP Per Capita Lag", "North America"]
```

```
y_name = ["Log RGDP Per Capita"]
```



```
Data
                          Coefficient        SE      t-stat  p-value signficance
y = Log RGDP Per Capita
EFW                          0.009240  0.002062    4.480037  0.00001         ***
Log RGDP Per Capita Lag      0.989891  0.001575  628.654269  0.00000         ***
North America               -0.010255  0.005034   -2.037096  0.04176           *
Constant                     0.060938  0.011515    5.291889  0.00000         ***
           Estimation Statistics
r**2                     0.997011
Adj. r**2                0.997007
f-stat              242384.675800
Est Var                  0.004596
MSE                      0.067791
SSE                     10.018495
SSR                   3341.737987
SST                   3351.756482
Obs.                  2184.000000
DOF                   2180.000000

Diff Data
                          Coefficient        SE     t-stat  p-value signficance
y = Log RGDP Per Capita
EFW                         -0.000122  0.001496  -0.081414  0.93512
Log RGDP Per Capita Lag      0.260629  0.020231  12.882443  0.00000         ***
North America               -0.003797  0.004860  -0.781387  0.43467
Constant                     0.024420  0.010205   2.392932  0.01680           *
           Estimation Statistics
r**2                     0.076250
Adj. r**2                0.074884
f-stat                  55.799745
Est Var                  0.004063
MSE                      0.063742
SSE                      8.239832
SSR                      0.680149
SST                      8.919981
Obs.                  2032.000000
DOF                   2028.000000
```

Neither of regression yield an estimate for the indicator variable that is 1) statistically significant or 2) that significantly improves the goodness-of-fit (r^2). In fact, the f-statistic and adjusted r-squared values have both fallen for the second set of regressions. It appears that inclusion of a control for North American countries does not improve the regression.

Panel Regression

The indicator variable plays a key role in a very popular regression within economics: the panel or (fixed effects) regression. A panel regression is an OLS regression that includes an indicator variable for certain fixed attributes. Conventiently, the panel regression is often used to control for effects between different political units – i.e., cities, states, nations, etc.... Using the data from the previous examples, we will run a panel regression with fixed effects for each nation. We regress the data over time, holding constant a level effect provided by the unique indicator variable associated with each nation. The indicator variables that drive results in the panel regression adjust the y-intercept indicated by the constant β_0 with a unique adjustment for each state.

We can accommodate a Panel Regression by making a few additions and edits to the *regress()* method that we have already built. The first step for creating a panel regression will be to add the *create_indicator_variable()* function that we created earlier. Be sure to add *self* to the terms that are passed to this method within the *Regression* class.

```python
#regression.py
import pandas as pd
import numpy as np
import copy
from stats import *
from scipy.stats import t, f

class Regression:
    def __init__(self):
        self.stats = Stats()
        self.reg_history = {}

    def OLS(self, reg_name, data, y_name, beta_names, min_val = 0,
            max_val = None, constant = True):
        # min_val and max_val set index range by index number
        self.min_val = min_val
        if max_val != None:
            self.max_val = max_val
        else:
            self.max_val = len(data)
        self.reg_name = reg_name
        # enodogenous variable name
        self.y_name = y_name
        # names of X variables
        self.beta_names = copy.copy(beta_names)
        #make a copy of the data that is passed to OLS
        self.data = data.copy()
        # if the OLS regression has a constant, add column of 1s to data
        if constant:
            self.add_constant()
        self.build_matrices()
        self.estimate_betas_and_yhat()
        self.calculate_regression_stats()
        self.save_output()

    def create_indicator_variable(self,data, indicator_name, index_name,
                                  target_index_list):
        # Prepare column with name of indicator variable
        data[indicator_name] = 0
        # for each index whose name matches an entry in target_index_list
        # a value of 1 will be recorded
        for index in target_index_list:
            data[indicator_name].loc[data.index.get_level_values(\
                index_name) == index] = 1
```

We will use the *create_indicator_variable()* method to create an indicator variable for every unique id in the index column labeled “ISO_Codes”. Each unique ISO Code represents a particular country, thus we will be creating one indicator variable for every country.

Since the index includes includes both “Year” and “ISO_Code”, we must select which type of fixed effect the regression will employ. We create the regression to allow for entity *or* time fixed effects in a single regression, but not both. Selection of both will lead lead the system to exit.

```python
#regression.py
import pandas as pd
import copy
import numpy as np
from scipy.stats import t
import matplotlib.pyplot as plt
from stats import Stats
import sys
. . .
    def panel_regression(self, reg_name, data, y_name, X_names, min_val = 0,
                         max_val = None, entity = False, time = False,
                         constant = True, ):
        if (entity and time) and  (not entity and not time):
            print("Choose time OR entity for panel regression.")
            sys.exit()
```

If you pass *entity = True* and *time = True*, the console will return the following:

Output:

```
SystemExit

C:\ProgramData\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py:3304:
UserWarning: To exit: use 'exit', 'quit', or Ctrl-D.

  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

Next, the *panel_regression()* method must determine the index to which it will refer for creating indicator variables. First, we identify which index column houses *datetime* data, and which houses *entity_data* by using *isinstance(obj, type)*. Using a for loop, we can check the identity of both index columns. We save their locations as *date_level* and *entity_level*. We save the names of these columns as *date_index_name* and *entity_index_name*.

```python
        #identify which index column holds dates, which holds entities
        for i in range(len(data.index.levels)):
            if isinstance(data.index.levels[i], pd.DatetimeIndex):
                date_level = i
                date_index_name = data.index.names[date_level]
            else:
                entity_level = i
                entity_index_name = data.index.names[entity_level]
```

Once the type of each column has been determined, save the name of the desired index as *index_name*. We can create indicator variables for each state (or time period if that is preferred) using the *create_indicator_variable()* method. Before , extract the list of entities names, reduce them to the unique *set*, transform the set into a list, and sort the list. Remove the last *indicator* in *indicators* by using *.pop()*. The nation missing an indicator variable will implicitly depend upon β_0. This value will serve as the anchor for the underlying regression that the indicator variables marginally shift for each state. Next, cycle through the list of *indicators* to create an indicator variable for each entity. We save the names of the indicator variables with the variables in *X*_names as *X_and_indicator_names* so that they can be referenced in the regression.

```
        #save name of selected index
        index_name = entity_index_name if entity else date_index_name
        #reduce list to unique elements and sort
        self.indicator_names = list(set(data.index.get_level_values(index_name)))
        self.indicator_names = sorted(self.indicator_names)
        self.indicator_names.pop()

        for indicator in self.indicators:
            self.create_indicator_variable(data, indicator, index_name,
                                           [indicator])
        X_and_indicator_names = X_names + self.indicators
```

All that is left is to run the regression. The key here is to include the names of the indicator variables in the regression but not include them in the results. To do this, we set *self.X_names* to include the names passed to *panel_regression()* and "Constant". The dataframe owned by the *Regression* class, *data*, is saved as to only include the variables indicated in self.*X_names*. Likewise, the estimated beta values for the indicator variables are removed by passing *self.X_names* to the index of *self.estimates.*

```

        self.OLS(reg_name = reg_name, data = data, y_name = y_name,
                X_names = X_and_indicator_names, min_val = min_val,
                max_val = max_val, constant = constant)
        self.X_names = X_names + ["Constant"]
        self.data = self.data[self.X_names]
        self.estimates = self.estimates.loc[self.X_names]

    def create_indicator_variable(self, data, indicator_name, index_name,
                                  target_index_list):
        # Prepare column with name of indicator variable
        data[indicator_name] = 0
        # for each index whose name matches an entry in target_index_list
        # a value of 1 will be recorded
        for index in target_index_list:
            data[indicator_name].loc[(data.index.get_level_values(\
                index_name)== index)] = 1
```

Finally, create a script modeled from *indicatorRegression.py*. We've removed any elements relating to the indicator variables from the previous script and saved the result as *panelRegression.py*. You may find copying this script manually to be easier than marginally editing the previous script. Note that we have removed "North America" from *X_names* and that *regress()* has been replaced by *panel_regression()*.

```
#panelRegression.py
import pandas as pd
import datetime
import numpy as np
import regression

# Import data with "ISO_Code" and "Year" as index columns
data = pd.DataFrame.from_csv("fraserDataWithRGDPPC.csv",
         index_col=[0,1], parse_dates = True)
data["RGDP Per Capita Lag"] = data.groupby(level=0)["RGDP Per Capita"].shift(-1)
data = data[data.index.get_level_values("Year") > datetime.datetime(1999,1,1)]

```

```python
for key in data:
    if "GDP" in key:
        data["Log " + key] = np.log(data[key])

# place dataframe in dictionary
# this will increase efficiency of managing differenced data
data_dict = {}
data_dict["Data"] = data
data_dict["Diff Data"] = data.groupby(level=0).diff(-1).dropna()

# prepare regression variables
X_names = ["EFW", "Log RGDP Per Capita Lag"]
y_name = ["Log RGDP Per Capita"]

for key in data_dict:
    #save instance of regression class
    reg = regression.Regression()
    # call panel_regression method
    reg.panel_regression(reg_name = key, data = data_dict[key].dropna(),
                y_name = y_name, X_names = X_names, entity = True,
                time = False)
    print(key + "\n", reg.estimates)
    print(reg.stats_DF)
    print()
```

Output:

```
Data
                           Coefficient        SE    t-stats  p-value p-rating
y = Log RGDP Per Capita
SUMMARY INDEX                 0.047439  0.007641   6.208501      0.0      ***
Log RGDP Per Capita Lag       0.825330  0.010592  77.920526      0.0      ***
Constant                      1.062765  0.080821  13.149673      0.0      ***
         Estimation Statistics
r**2                    0.998262
f-stat               5132.307544
Est Var                 0.002222
MSE                     0.047141
SSE                     2.164484
SSR                  1243.181690
SST                  1245.346174

Diff Data
                           Coefficient        SE   t-stats  p-value p-rating
y = Log RGDP Per Capita
SUMMARY INDEX                 0.033907  0.013228  2.563242  0.01054        *
Log RGDP Per Capita Lag       0.069807  0.032862  2.124272  0.03393        *
Constant                     -0.005754  0.016688 -0.344784  0.73034
         Estimation Statistics
r**2                    0.175910
f-stat                  1.753565
Est Var                 0.002907
MSE                     0.053921
SSE                     2.555676
SSR                     0.545535
SST                     3.101211
```

The inclusion of entity fixed effects has improved the overall fit, though it has also diminished both the estimate of statistical significance (indicated by the p-value) and economic significance (beta estimate) of a nation’s economic freedom on its real GDP per capita.

We want to be sure that using a panel regression actually improved the estimates. To do this, we run a joint f-test, as in the previous chapter. We will provide a unique name for

```python
#panelRegression.py
. . .
for key in data_dict:
    # call OLS and Panel for comparison
    data = data_dict[key]
    reg.OLS(reg_name = key, data = data.dropna(),
            y_name = y_name, beta_names = X_names)
    print(key, reg.estimates, sep = "\n")
    print(reg.stats_DF)
    panel_name = key + " panel"
    reg.panel_regression(reg_name = panel_name, data = data.dropna(),
            y_name = y_name, X_names = X_names, entity = True, time = False)
    print(key, reg.estimates, sep = "\n")
    print(reg.stats_DF)
    joint_f_test = reg.joint_f_test(key, key + " panel")
    joint_f_test.to_csv(key + " panel comparison.csv")
    print()
```

The joint f-test for each pair of regressions is saved as csvs named “Data panel comparison.csv” and “Diff Data panel comparison.csv”. The results are shown below, respectively.

AGO = ALB = ARE = ARG = ARM	
f-stat	5.013193
p-value	1.11E-16
dfn	151
dfd	2029

AGO = ALB = ARE = ARG = ARM	
f-stat	1.657733
p-value	2.89E-06
dfn	148
dfd	1880

Since both joint f-test generate statistics whose associated p-value especially small ($p < .001$), it appears that the panel regression is an improvement in both cases.

Regression Libraries in Python

While any competent data scientist should be able to build functions on the fly, they should also be familiar with the libraries that are already provided in *python*. This chapter will introduce the *statsmodels*.

Regession in *statsmodels*

We can compare results of our regression to results of a regression from the popular *statsmodels* library. The setup for the regression is straightforward. Create an endogenous variable y and set of exogenous variables X. Run the regression and print the results.

```python
#statsmodelsOLS.py
import pandas as pd
import statsmodels.api as sm

data = pd.read_csv("cleanedEconFreedomData.csv", index_col = ["Country Name"])
y_var = ["5 Year GDP Growth Rate (%)"]
x_vars = ["Gov't Expenditure % of GDP ", "2017 Score",
          "Population (Millions)", "Property Rights", "Business Freedom"]

y = data[y_var]
X = data[x_vars]
X["Constant"] = 1
results = sm.OLS(y, X).fit()
```

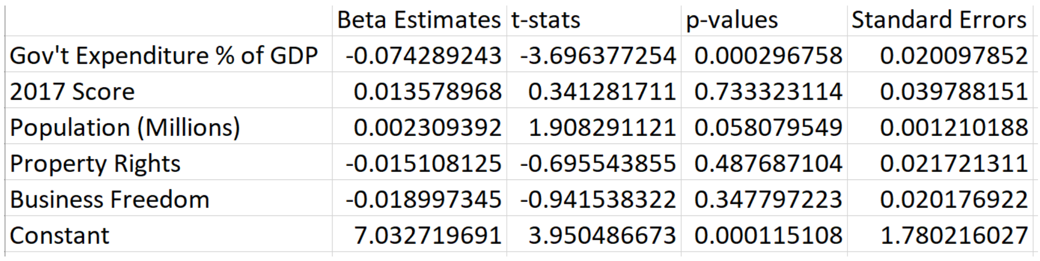
The command sm.OLS(y, X) prepares the regression. Adding *.fit()* runs the regression and saves the results. Statistics can be called individually or in a a table that gathers all output for the regression.

You may want to present only the estimated beta coefficients, t-statistics, p-values, and standard errors. *statsmodels* provides commands to call an array that holds estimates for each variable. These can conveniently be called saved as a dataframe in a few steps. We will call these values, save them in a dictionary, then transform the dictionary into a dataframe.

```python
. . .
betaEstimates = results.params
tStats = results.tvalues
pValues =  results.pvalues
stdErrors = results.bse

resultsDict = {"Beta Estimates": betaParams,
               "t-stats": tStats,
               "p-values": pValues,
               "Standard Errors": stdErrors}
resultsDF = pd.DataFrame(resultsDict)
```

Since we have identified the name of each set of results – "Beta Estimates", "t-stats", etc… – the dataframe is easy to interpret. To view in whole, we can output to a csv.

```python
resultsDF.to_csv(“statsmodelsOLSResults.csv”)
```



	Beta Estimates	t-stats	p-values	Standard Errors
Gov't Expenditure % of GDP	-0.074289243	-3.696377254	0.000296758	0.020097852
2017 Score	0.013578968	0.341281711	0.733323114	0.039788151
Population (Millions)	0.002309392	1.908291121	0.058079549	0.001210188
Property Rights	-0.015108125	-0.695543855	0.487687104	0.021721311
Business Freedom	-0.018997345	-0.941538322	0.347797223	0.020176922
Constant	7.032719691	3.950486673	0.000115108	1.780216027

Alternately, we can call the full table of results:

```python
. . .
```

```
OLSSummary = results.summary()
```

```
                                 OLS Regression Results
==============================================================================
Dep. Variable:     5 Year GDP Growth Rate (%)   R-squared:                       0.210
Model:                                    OLS   Adj. R-squared:                  0.186
Method:                         Least Squares   F-statistic:                     8.799
Date:                        Sun, 10 Mar 2019   Prob (F-statistic):           2.00e-07
Time:                                21:38:28   Log-Likelihood:                -382.26
No. Observations:                         172   AIC:                             776.5
Df Residuals:                             166   BIC:                             795.4
Df Model:                                   5
Covariance Type:                    nonrobust
===============================================================================================
                                 coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------------------
Gov't Expenditure % of GDP    -0.0743      0.020     -3.696      0.000      -0.114      -0.035
2017 Score                     0.0136      0.040      0.341      0.733      -0.065       0.092
Population (Millions)          0.0023      0.001      1.908      0.058      -8e-05       0.005
Property Rights               -0.0151      0.022     -0.696      0.488      -0.058       0.028
Business Freedom              -0.0190      0.020     -0.942      0.348      -0.059       0.021
Constant                       7.0327      1.780      3.950      0.000       3.518      10.548
==============================================================================
Omnibus:                       21.468   Durbin-Watson:                   1.849
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               58.102
Skew:                          -0.456   Prob(JB):                     2.42e-13
Kurtosis:                       5.698   Cond. No.                     1.62e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.62e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

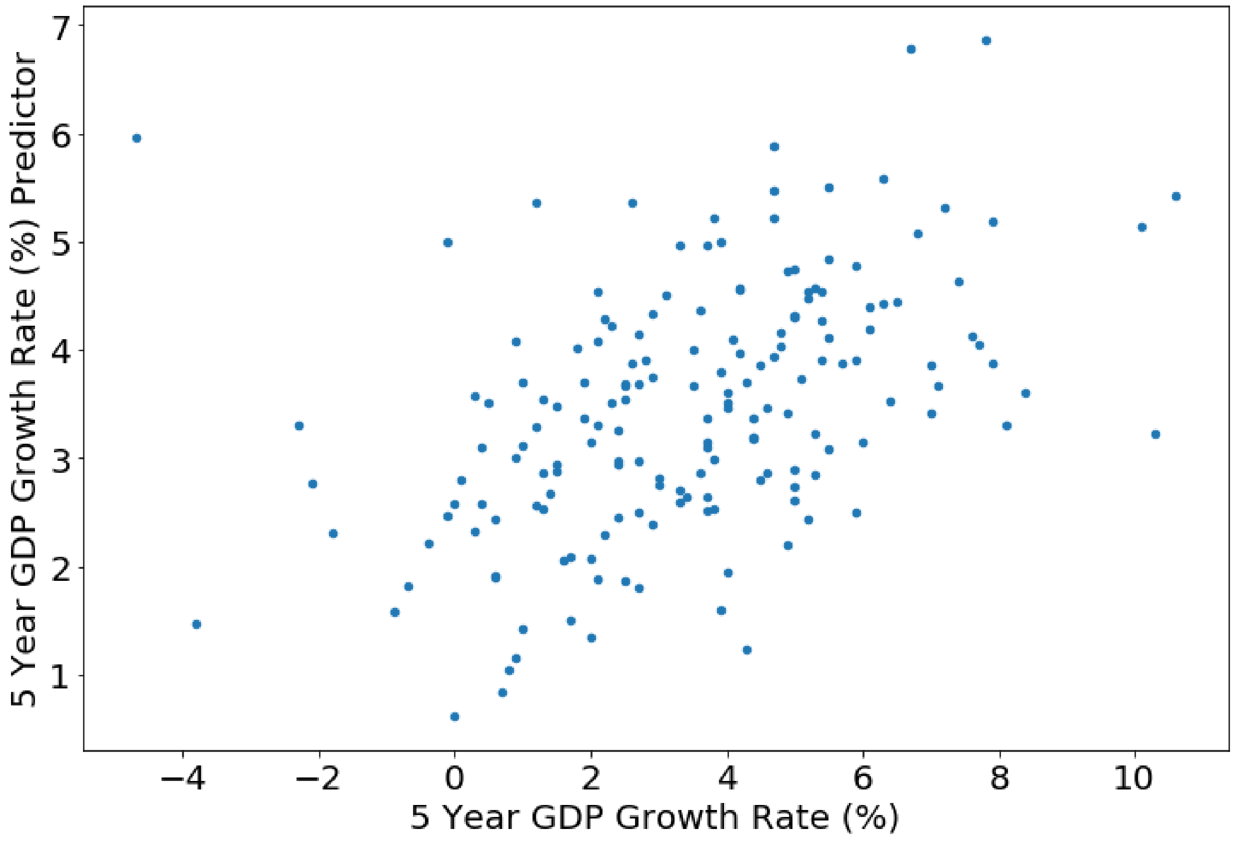
A brief comparison of these results with the ones generated by the model you created show that they are precisely the same. The OLS model from *statsmodels* provides a handful of other statistics including confidence intervals, adjusted r-squared, and a Durbin-Watson statistic, among others.[2]

Finally, as with the OLS model that we built in Chapter 7, you may call the predictor that is estimated by the regression. Save the predictor in the dataframe holding the data that was imported at the earlier in the script. Then, we can plot a scatter plot to compare the predicted y-value with the observed value. To adjust the size of the plot and the fontsize of the axis titles, we will need to import *matplotlib.pyplot*. By calling *plt.subplots* you can assign the plot from pandas to the figure using *data.plot.scatter(. . ., ax = ax).*

```
#statsmodelsOLS.py
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

. . .

```

[2] For a more extensive presentation see: https://www.statsmodels.org/dev/examples/notebooks/generated/ols.html

```
predictor = results.predict()
data[y_var[0] + " Predictor"] = predictor
fig, ax = plt.subplots(figsize = (12,8))
plt.rcParams.update({"font.size": 18})
data.plot.scatter(x = y_var[0], y = y_var[0] + " Predictor", ax = ax)
```



Panel OLS Regression

Double indexed data is especially useful for panel regressions. A panel regression separates observations according to different categories. This allows the panel regression to identify fixed effects. Fixed effects can shift the level of a given category of identity. For example, observations across different political regions may experience entity fixed effects that persist over time. Thus, each political regiions can be assigned an entity fixed effect that adjusts the constant of the regression equation for that particular entity.

To begin, you will need to install *linearmodels*[3] using the command line.

```
pip install linearmodels
```

[3] For more information, visit: https://bashtage.github.io/linearmodels/doc/panel/examples/examples.html

Once linear models is installed, we can import the *PanelOLS* model to run a panel regression. As in the previous example, import the *fraserDataWithRGDPPC.csv*. We will run a panel regression checking for the effects of *Sound Money*, *Government Consumption*, and *Economic Freedom Index Scores* (*SUMMARY INDEX*) on *Real GDP Per Capita*. Select this subset of data as an object named *reg_data*.

```python
import pandas as pd
from linearmodels import PanelOLS

#import data
data = pd.DataFrame.from_csv("fraserDataWithRGDPPC.csv", index_col=[0,1],
                             parse_dates = True)

# Panel OLS
# save dataframe with only the variables that will be used in regression
# and drop any observations that are missing any one of these values
reg_data = data[["RGDP Per Capita", "Sound Money", "Government Consumption",
                 "SUMMARY INDEX"]].dropna()
```

Next, identify the dependent variable in a list named *y_name*, and the dependent variables in a list named *x_name*. Then save the data of the dependent variable as *y* and data of the independent variables as *X.*

```python
. . .
# RGDP Per Capita is the dependent variable
y_name = ["RGDP Per Capita"]
# Sound Money, Government Consumption, and SUMMARY INDEX are our indep vars
x_names = ["Sound Money", "Government Consumption", "SUMMARY INDEX"]
# save dependent and independent variables as their own objects
y = reg_data[y_name]
X = reg_data[x_names]
```

Next, prepare the the regression using *PanelOLS* and save this as an object named *mod*. We will be using entity fixed effects, so be sure to market *entity_effects* as True. Next, we fit a predictor using robust standard errors by setting *cov_type= "clustered"* and *cluster_entity=True*.

```python
. . .
# prepare panelOLS controlling for entity fixed effects
mod = PanelOLS(y, X, entity_effects=True, time_effects=False)
# estimate with robust errors
res = mod.fit(cov_type='clustered', cluster_entity=True)
```

We can now observe the results of the regression by entering *res* in the console:

```
res
```

Output:

```
                          PanelOLS Estimation Summary
================================================================================
Dep. Variable:        RGDP Per Capita   R-squared:                        0.1848
Estimator:                   PanelOLS   R-squared (Between):              0.5639
No. Observations:                2862   R-squared (Within):               0.1848
Date:                Thu, Aug 22 2019   R-squared (Overall):              0.5915
Time:                        12:27:36   Log-likelihood                -2.868e+04
Cov. Estimator:             Clustered
                                        F-statistic:                      204.52
Entities:                         162   P-value                           0.0000
Avg Obs:                       17.667   Distribution:                  F(3,2706)
Min Obs:                       0.0000
Max Obs:                       23.000   F-statistic (robust):             19.811
                                        P-value                           0.0000
Time periods:                      23   Distribution:                  F(3,2706)
Avg Obs:                       124.43
Min Obs:                       52.000
Max Obs:                       153.00

                                 Parameter Estimates
==========================================================================================
                        Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------------------
Sound Money                143.60     223.41     0.6428     0.5204     -294.47      581.67
Government Consumption    -1429.4     356.08    -4.0142     0.0001     -2127.6     -731.16
SUMMARY INDEX              2814.0     525.61     5.3538     0.0000      1783.4      3844.6
==========================================================================================

F-test for Poolability: 74.684
P-value: 0.0000
Distribution: F(152,2706)

Included effects: Entity
PanelEffectsResults, id: 0x1c4e07ba240
```

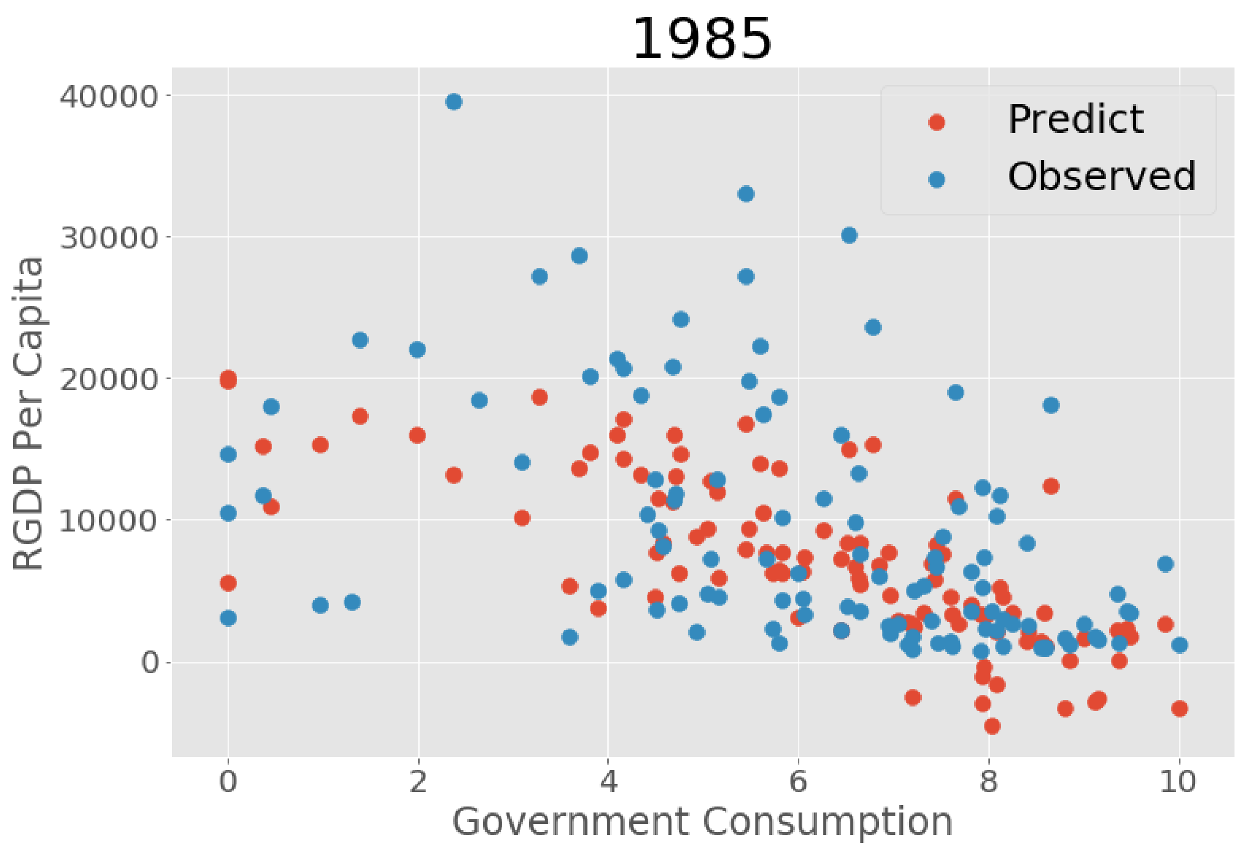
The regression seems to provide a relatively good fit for the data. We have certainly left variables out that would be important to include, such as . Our results suggest that *Economic Freedom* (*SUMMARY INDEX*) tends to positively impact real GDP per capita while *Government Consumption* negatively impacts real GDP per capita.

We can now save the predictor generated by the regression and record this in the original dataframe. Notice that for missing values, Nan is recorded. As with the earlier merging of data, the double index makes it possible to efficiently save an incomplete column of data in the dataframe.

```
#save predictor and place values in the original dataframe named data
predictor = res.predict()
predict_name = y_name[0] + " Predict"
data[predict_name] = predictor
```

As with the earlier example, it is easiest to view the new data by saving the dataframe as a csv using *data.to_csv()*.

As with the OLS regression covered in the beginning of the chapter, we can also plot the prediction



Final Project: Create 4 dimensional scatter plots comparing predicted values from panel regression with observed values

Loop Scatter Plots with Double Index

Next we will import the dataset

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from matplotlib.backends.backend_pdf import PdfPages

def four_dim_scatter(data, year, x_name, y_name, c_name, s_name, smin=1,
                     smax=10, figsize = (16, 10), index_name = "Year",
                     color_bar = "Dark2",discrete_color = False, pp = None):

    # select a subset of the data
    data_year = data[(data.index.get_level_values(index_name) == year)][plot_vars]
    # remove any row with nan value
    data_year = data_year.dropna(thresh =len(data_year.columns))

    # Create a figure and axis; choose size of the figure
    fig, ax = plt.subplots(figsize=figsize)

    # set plot values before hand, easier to interpret
    x = data_year[x_name]
    y = data_year[y_name]
    c = data_year[c_name]
    s = data_year[s_name]

    #chooses color schema, number of colors
    if discrete_color:
        color_divisions = len(set(data[c_name].dropna()))
        cmap = plt.cm.get_cmap(color_bar, color_divisions)
    else:
        cmap = plt.cm.get_cmap(color_bar)
    # use ax to plot scatter()

    scatter = scatter_legend(ax, x, y, s, c, cmap, s_name, smin, smax)

    setup_cbar(scatter, color_divisions, c_name)

    # set tick line length to 0
    ax.tick_params(axis=u'both', which=u'both',length=0)
    # set axis value labels fontsize
    plt.xticks(fontsize = 20)
    plt.yticks(fontsize = 20)
    # set axis range
    plt.xlabel(x_name,fontsize = 20)
    plt.ylabel(y_name, fontsize = 20)
    plt.xlim(3,10)
    plt.ylim(3,10)

    # Make title year for each scatter plot
    plt.title(str(year)[:4], fontsize = 30)
    plt.show()
    if pp != None:
        pp.savefig(fig, bbox_inches = "tight")
    plt.close()

def scatter_legend(ax, x, y, s, c, cmap, s_name, smin, smax):
    scatter = ax.scatter(x=x,y=y,s=s,c=c,cmap=cmap)

    # create legend for scatter dot sizes
    # first build blank plots with desired sizes for legend
    smid = (smin + smax)  / 2
    gmin = plt.scatter([],[], s=10, marker='o', color='#555555')
    gmid = plt.scatter([],[], s=smid / 4, marker='o', color='#555555')

    gmax = plt.scatter([],[], s=smax / 4, marker='o', color='#555555')
    # create legend for plot size
    ax.legend((gmin,gmid,gmax),
       ("", s_name, ""),
       # bbox_to_anchor and loc set position of legend
       bbox_to_anchor=(0, -0.17),
       loc='lower left',
       fontsize=12)

    return scatter

def setup_cbar(scatter, color_divisions, c_name):
    # include colorbar
    cbar = plt.colorbar(scatter)
    # this centers the number bar value labels
    # change some of the values passed to see what happens....
    tick_locs = (np.arange(1, color_divisions+1) + .82)\
        *(color_divisions-1)/color_divisions
    cbar.set_ticks(tick_locs)
    # choose numbers 1 through 4 as colorbar labels
    cbar.set_ticklabels(np.arange(1,color_divisions+1))
    cbar.ax.tick_params(labelsize=20)
    # add general label to colorbar
    cbar.set_label(c_name, size=20)

#changes background of plot
plt.style.use('ggplot')

#import data
data = pd.DataFrame.from_csv("fraserDataWithRGDPPC.csv", index_col=[0,1])

# create list of each index set from multi index
years = list(sorted(set(data.index.get_level_values('Year'))))
country = list(sorted(set(data.index.get_level_values('ISO_Code'))))
#choose variables that will be plotted for each year in scatter
plot_vars = ["Sound Money", "Government Consumption",
             "RGDP Per Capita","Quartile"]

       # Normalize income so that 1000 represents the maximum value of RGDP Per Capita
       # This will allow dot to be easily adjusted
       data["RGDP Per Capita"] = data["RGDP Per Capita"] /\
           max(data["RGDP Per Capita"]) * 1000
       smin = min(data["RGDP Per Capita"])
       smax = max(data["RGDP Per Capita"])

       pp = PdfPages("Scatter Plots.pdf")
       x = "Sound Money"
       y ="Government Consumption"
       c = "Quartile"
       s = "RGDP Per Capita"

       for year in years:
           four_dim_scatter(data, year, x, y, c, s,smin, smax, index_name = "Year",
                            discrete_color=True, pp=pp)
       pp.close()
```