DeLong's Macroeconomics

Introduction: Python and Economics

Due ???? via upload to ???

J. Bradford DeLong

You should have gotten to this point vis this link: http://datahub.berkeley.edu/user-redirect/interact?account=braddelong&repo=LS2019&branch=master&path=Introduction-Python_and_Economics.ipynb (http://datahub.berkeley.edu/user-redirect/interact?account=braddelong&repo=LS2019&branch=master&path=Introduction-Python_and_Economics.ipynb)

Welcome to Economics 101B: Macroeconomic Theory! This introductory notebook will familiarize you with some of the basic strategies for data analysis that will be useful to you throughout the course. It will cover an overview of our software, an introduction to programming, and some economics.

Table of Contents

1. Computing Environment
2. Introduction to Coding Concepts
   A. Python Basics
   B. Pandas
   C. Visualization
3. Math Review
4. Economics Review: Measuring the Economy
5. Economics Review: Economic Thought

Our Computing Environment: The Jupyter Notebook

This webpage is called a Jupyter notebook. A notebook is a place to write programs and view their results.

Markdown Text Cells

In a notebook, each rectangle containing text or code is called a *cell*.

Text cells (like this one) can be edited by double-clicking on them. They're written in a simple format called Markdown (http://daringfireball.net/projects/markdown/syntax) to add formatting and section headings. You don't need to learn Markdown, but you might want to.

After you edit a text cell, click the "run cell" button at the top that looks like ▶| to confirm any changes. (Try not to delete the instructions of the lab.)

Understanding Check 1 This paragraph is in its own text cell. Try editing it so that this sentence is the last sentence in the paragraph, and then click the "run cell" ▶| button . This sentence, for example, should be deleted. So should this one.

Python Code Cells

Other cells contain code in the Python 3 language. Running a code cell will execute all of the code it contains.

To run the code in a code cell, first click on that cell to activate it. It'll be highlighted with a little green or blue rectangle. Next, either press ▶| or hold down the `shift` key and press `return` or `enter` .

Try running this cell:

In [1]:
```
print("Hello, World!")
```

```
Hello, World!
```

And this one:

In [2]:
```
print("\N{WAVING HAND SIGN}, \N{EARTH GLOBE ASIA-AUSTRALIA}!")
```

```
👋, 🌏!
```

The fundamental building block of Python code is an expression. Cells can contain multiple lines with multiple expressions. When you run a cell, the lines of code are executed in the order in which they appear. Every `print` expression prints a line. Run the next cell and notice the order of the output.

In [6]:
```
print("First this line,")
print("then the whole \N{EARTH GLOBE ASIA-AUSTRALIA},")
print("and then this one,")
```

```
First this line,
then the whole 🌏,
and then this one,
```

In [3]:
```
print("First this line is printed,")
print("and then this one.")
```

```
First this line is printed,
and then this one.
```

Understanding Check 2 Change the cell above so that it prints out:

```
First this line,
then the whole 🌏,
and then this one.
```

Don't be scared if you see a "Kernel Restarting" message! Your data and work will still be saved. Once you see "Kernel Ready" in a light blue box on the top right of the notebook, you'll be ready to work again. You should rerun any cells with imports, variables, and loaded data.

Writing Jupyter notebooks

You can use Jupyter notebooks for your own projects or documents. When you make your own notebook, you'll need to create your own cells for text and code.

To add a cell, click the + button in the menu bar. It'll start out as a text cell. You can change it to a code cell by clicking inside it so it's highlighted, clicking the drop-down box next to the restart (⟳) button in the menu bar, and choosing "Code".

Understanding Check 3 Add a code cell below this one. Write code in it that prints out:

```
A whole new cell! ♪🌍♪
```

(That musical note symbol is like the Earth symbol. Its long-form name is `\N{EIGHTH NOTE}` .)

Run your cell to verify that it works.

In [8]:
```
print("A whole new cell! \N{EIGHTH NOTE}\N{EARTH GLOBE ASIA-AUSTRALIA}\N{EIGHTH NOTE}")
```

```
A whole new cell! ♪🌏♪
```

Errors

Python is a language, and like natural human languages, it has rules. It differs from natural language in two important ways:

1. The rules are *simple*. You can learn most of them in a few weeks and gain reasonable proficiency with the language in a semester.
2. The rules are *rigid*. If you're proficient in a natural language, you can understand a non-proficient speaker, glossing over small mistakes. A computer running Python code is not smart enough to do that.

Whenever you write code, you'll make mistakes. When you run a code cell that has errors, Python will sometimes produce error messages to tell you what you did wrong.

Errors are okay; even experienced programmers make many errors. When you make an error, you just have to find the source of the problem, fix it, and move on.

We have made an error in the next cell. Run it and see what happens.

In [10]:
```
print("This line is missing something.")
```

```
This line is missing something.
```

You should see something like this (minus our annotations):

The last line of the error output attempts to tell you what went wrong. The *syntax* of a language is its structure, and this `SyntaxError` tells you that you have created an illegal structure. " `EOF` " means "end of file," so the message is saying Python expected you to write something more (in this case, a right parenthesis) before finishing the cell.

There's a lot of terminology in programming languages, but you don't need to know it all in order to program effectively. If you see a cryptic message like this, you can often get by without deciphering it. (Of course, if you're frustrated, feel free to ask a friend or post on the class Piazza.)

Understanding Check 4 Try to fix the code above so that you can run the cell and see the intended message instead of an error.

Now that you are comfortable with our computing environment, we are going to be moving into more of the fundamentals of Python, but first, run the cell below to ensure all the libraries needed for this notebook are installed.

In [11]:
```
!pip install numpy
!pip install pandas
!pip install matplotlib
```

```
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020.
 Please upgrade your Python as Python 2.7 won't be maintained after that dat
e. A future version of pip will drop support for Python 2.7.
Requirement already satisfied: numpy in /usr/local/lib/python2.7/dist-package
s (1.14.3)
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020.
 Please upgrade your Python as Python 2.7 won't be maintained after that dat
e. A future version of pip will drop support for Python 2.7.
Requirement already satisfied: pandas in /usr/local/lib/python2.7/dist-packag
es (0.23.4)
Requirement already satisfied: pytz>=2011k in /usr/lib/python2.7/dist-package
s (from pandas) (2018.3)
Requirement already satisfied: numpy>=1.9.0 in /usr/local/lib/python2.7/dist-
packages (from pandas) (1.14.3)
Requirement already satisfied: python-dateutil>=2.5.0 in /usr/lib/python2.7/d
ist-packages (from pandas) (2.6.1)
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020.
 Please upgrade your Python as Python 2.7 won't be maintained after that dat
e. A future version of pip will drop support for Python 2.7.
Requirement already satisfied: matplotlib in /usr/lib/python2.7/dist-packages
(2.1.1)
```

Introduction to Programming Concepts

Part 1: Python Basics

Before getting into the more advanced analysis techniques that will be required in this course, we need to cover a few of the foundational elements of programming in Python.

A. Expressions

The departure point for all programming is the concept of the **expression**. An expression is a combination of variables, operators, and other Python elements that the language interprets and acts upon. Expressions act as a set of instructions to be fed through the interpreter, with the goal of generating specific outcomes. See below for some examples of basic expressions.

In [12]:
```
# Examples of expressions:

#addition
print(2 + 2)

#string concatenation
print('me' + ' and I')

#you can print a number with a string if you cast it
print("me" + str(2))

#exponents
print(12 ** 2)
```

```
4
me and I
me2
144
```

In [0]:
```
You will notice that only the last line in a cell gets printed out. If you wan
t to see the values of previous expressions, you need to call `print` on that
expression. Try adding `print` statements to some of the above expressions to
get them to display.

 
```

B. Variables

In the example below, `a` and `b` are Python objects known as **variables**. We are giving an object (in this case, an `integer` and a `float`, two Python data types) a name that we can store for later use. To use that value, we can simply type the name that we stored the value as. Variables are stored within the notebook's environment, meaning stored variable values carry over from cell to cell.

In [14]:
```
a = 4
b = 10/5
```

Notice that when you create a variable, unlike what you previously saw with the expressions, it does not print anything out.

In [15]:
```
# Notice that 'a' retains its value.
print(a)
a + b
```

```
4
```

Out[15]:
```
6.0
```

Question 1: Variables

See if you can write a series of expressions that creates two new variables called **x** and **y** and assigns them values of **10.5** and **7.2**. Then assign their product to the variable **combo** and print it.

In [16]:
```
# Fill in the missing lines to complete the expressions.



x = 10.5
y = 7.2
combo = x * y

print(combo)
```

```
75.60000000000001
```

Running the cell below will give you some feed back on your responses. Though the OK tests are not always comprehensive (passing all of the tests does not guarantee full credit for questions), they give you a pretty good indication as to whether or not you're on track.

C. Lists

The next topic is particularly useful in the kind of data manipulation that you will see throughout 101B. The following few cells will introduce the concept of **lists** (and their counterpart, `numpy arrays`). Read through the following cell to understand the basic structure of a list.

A list is an ordered collection of objects. They allow us to store and access groups of variables and other objects for easy access and analysis. Check out this documentation (https://www.tutorialspoint.com/python/python_lists.htm) for an in-depth look at the capabilities of lists.

To initialize a list, you use brackets. Putting objects separated by commas in between the brackets will add them to the list.

In [18]:
```
# an empty list
lst = []
print(lst)

# reassigning our empty list to a new list
lst = [1, 3, 6, 'lists', 'are' 'fun', 4]
print(lst)
```

```
[]
[1, 3, 6, 'lists', 'arefun', 4]
```

To access a value in the list, put the index of the item you wish to access in brackets following the variable that stores the list. Lists in Python are zero-indexed, so the indicies for `lst` are 0, 1, 2, 3, 4, 5, and 6.

In [19]:
```
# Elements are selected like this:
example = lst[2]

# The above line selects the 3rd element of lst (list indices are 0-offset) an
d sets it to a variable named example.
print(example)
```

```
6
```

It is important to note that when you store a list to a variable, you are actually storing the **pointer** to the list. That means if you assign your list to another variable, and you change the elements in your other variable, then you are changing the same data as in the original list.

In [20]:
```
a = [1,2,3] #original list
b = a #b now points to list a
b[0] = 4
print(a[0]) #return 4 since we modified the first element of the list pointed
 to by a and b
```

```
4
```

Slicing lists

As you can see from above, lists do not have to be made up of elements of the same kind. Indices do not have to be taken one at a time, either. Instead, we can take a slice of indices and return the elements at those indices as a separate list.

In [21]:
```
### This line will store the first (inclusive) through fourth (exclusive) elem
ents of lst as a new list called lst_2:
lst_2 = lst[1:4]

lst_2
```

Out[21]:
```
[3, 6, 'lists']
```

Question 2: Lists

Build a list of length 10 containing whatever elements you'd like. Then, slice it into a new list of length five using a index slicing. Finally, assign the last element in your sliced list to the given variable and print it.

In [28]:
```
### Fill in the ellipses to complete the question.
lst = [1,3,5,6,7,9,"HELLO","PYTHON",0,11]
lst_2 = lst[0:5]
a = lst_2[4]
print(a)
```

```
7
```

Lists can also be operated on with a few built-in analysis functions. These include `min` and `max`, among others. Lists can also be concatenated together. Find some examples below.

In [29]:
```
# A list containing six integers.
a_list = [1, 6, 4, 8, 13, 2]

# Another list containing six integers.
b_list = [4, 5, 2, 14, 9, 11]

print('Max of a_list:', max(a_list))
print('Min of b_list:', min(a_list))

# Concatenate a_list and b_list:
c_list = a_list + b_list
print('Concatenated:', c_list)
```

```
Max of a_list: 13
Min of b_list: 1
Concatenated: [1, 6, 4, 8, 13, 2, 4, 5, 2, 14, 9, 11]
```

D. Numpy Arrays

Closely related to the concept of a list is the array, a nested sequence of elements that is structurally identical to a list. Arrays, however, can be operated on arithmetically with much more versatility than regular lists. For the purpose of later data manipulation, we'll access arrays through Numpy (https://docs.scipy.org/doc/numpy/reference/routines.html), which will require an import statement.

Now run the next cell to import the numpy library into your notebook, and examine how numpy arrays can be used.

In [30]:
```
import numpy as np
```

In [31]:
```
# Initialize an array of integers 0 through 9.
example_array = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# This can also be accomplished using np.arange
example_array_2 = np.arange(10)
print('Undoubled Array:')
print(example_array_2)

# Double the values in example_array and print the new array.
double_array = example_array*2
print('Doubled Array:')
print(double_array)
```

```
Undoubled Array:
[0 1 2 3 4 5 6 7 8 9]
Doubled Array:
[ 0  2  4  6  8 10 12 14 16 18]
```

This behavior differs from that of a list. See below what happens if you multiply a list.

In [32]:
```
example_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
example_list * 2
```

Out[32]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that instead of multiplying each of the elements by two, multiplying a list and a number returns that many copies of that list. This is the reason that we will sometimes use Numpy over lists. Other mathematical operations have interesting behaviors with lists that you should explore on your own.

E. Looping

Loops (https://www.tutorialspoint.com/python/python_loops.htm) are often useful in manipulating, iterating over, or transforming large lists and arrays. The first type we will discuss is the **for loop**. For loops are helpful in traversing a list and performing an action at each element. For example, the following code moves through every element in example_array, adds it to the previous element in example_array, and copies this sum to a new array.

In [33]:
```
new_list = []

for element in example_array:
    new_element = element + 5
    new_list.append(new_element)

new_list
```

Out[33]:
```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

The most important line in the above cell is the " `for element in...` " line. This statement sets the structure of our loop, instructing the machine to stop at every number in `example_array`, perform the indicated operations, and then move on. Once Python has stopped at every element in `example_array`, the loop is completed and the final line, which outputs `new_list`, is executed. It's important to note that "element" is an arbitrary variable name used to represent whichever index value the loop is currently operating on. We can change the variable name to whatever we want and achieve the same result, as long as we stay consistent. For example:

In [34]:
```
newer_list = []

for completely_arbitrary_name in example_array:
    newer_element = completely_arbitrary_name + 5
    newer_list.append(newer_element)

newer_list
```

Out[34]:
```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

For loops can also iterate over ranges of numerical values. If I wanted to alter `example_array` without copying it over to a new list, I would use a numerical iterator to access list indices rather than the elements themselves. This iterator, called `i`, would range from 0, the value of the first index, to 9, the value of the last. I can make sure of this by using the built-in `range` and `len` functions.

In [35]:
```
for i in range(len(example_array)):
    example_array[i] = example_array[i] + 5

example_array
```

Out[35]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Other Types of Loops

The **while loop** repeatedly performs operations until a conditional is no longer satisfied. A conditional is a boolean expression (https://en.wikipedia.org/wiki/Boolean_expression), that is an expression that evaluates to `True` or `False`.

In the below example, an array of integers 0 to 9 is generated. When the program enters the while loop on the subsequent line, it notices that the maximum value of the array is less than 50. Because of this, it adds 1 to the fifth element, as instructed. Once the instructions embedded in the loop are complete, the program refers back to the conditional. Again, the maximum value is less than 50. This process repeats until the the fifth element, now the maximum value of the array, is equal to 50, at which point the conditional is no longer true and the loop breaks.

In [36]:
```
while_array = np.arange(10)        # Generate our array of values

print('Before:', while_array)

while(max(while_array) < 50):      # Set our conditional
    while_array[4] += 1            # Add 1 to the fifth element if the conditi
onal is satisfied

print('After:', while_array)
```

```
Before: [0 1 2 3 4 5 6 7 8 9]
After: [ 0  1  2  3 50  5  6  7  8  9]
```

Question 3: Loops

In the following cell, partial steps to manipulate an array are included. You must fill in the blanks to accomplish the following:

1. Iterate over the entire array, checking if each element is a multiple of 5
2. If an element is not a multiple of 5, add 1 to it repeatedly until it is
3. Iterate back over the list and print each element.

> Hint: To check if an integer `x` is a multiple of `y`, use the modulus operator `%`. Typing `x % y` will return the remainder when `x` is divided by `y`. Therefore, `( x % y != 0 )` will return `True` when `y` **does not divide** `x`, and `False` when it does.

In [38]:
```
import numpy as np

# Make use of iterators, range, length, while loops, and indices to complete t
his question.
question_3 = np.array([12, 31, 50, 0, 22, 28, 19, 105, 44, 12, 77])
for i in range(11):
    while question_3[i] % 5 != 0:
        question_3[i] += 1
print(question_3)
```

```
[ 15  35  50   0  25  30  20 105  45  15  80]
```

F. Functions!

Functions are useful when you want to repeat a series of steps on multiple different objects, but don't want to type out the steps over and over again. Many functions are built into Python already; for example, you've already made use of `len()` to retrieve the number of elements in a list. You can also write your own functions, and at this point you already have the skills to do so.

Functions generally take a set of **parameters** (also called inputs), which define the objects they will use when they are run. For example, the `len()` function takes a list or array as its parameter, and returns the length of that list.

The following cell gives an example of an extremely simple function, called `add_two`, which takes as its parameter an integer and returns that integer with, you guessed it, 2 added to it.

In [39]:
```
# An adder function that adds 2 to the given n.
def add_two(n):
    return n + 2
```

In [40]:
```
add_two(5)
```

Out[40]: 7

Easy enough, right? Let's look at a function that takes two parameters, compares them somehow, and then returns a boolean value (`True` or `False`) depending on the comparison. The `is_multiple` function below takes as parameters an integer `m` and an integer `n`, checks if `m` is a multiple of `n`, and returns `True` if it is. Otherwise, it returns `False`.

`if` statements, just like `while` loops, are dependent on boolean expressions. If the conditional is `True`, then the following indented code block will be executed. If the conditional evaluates to `False`, then the code block will be skipped over. Read more about `if` statements here (https://www.tutorialspoint.com/python/python_if_else.htm).

In [41]:
```
def is_multiple(m, n):
    if (m % n == 0):
        return True
    else:
        return False
```

In [42]:
```
is_multiple(12, 4)
```

Out[42]: True

In [43]:
```
is_multiple(12, 7)
```

Out[43]: False

Sidenote: Another way to write `is_multiple` is below, think about why it works.

```
def is_multiple(m, n):
    return m % n == 0
```

Since functions are so easily replicable, we can include them in loops if we want. For instance, our `is_multiple` function can be used to check if a number is prime! See for yourself by testing some possible prime numbers in the cell below.

In [45]:
```
# Change possible_prime to any integer to test its primality
# NOTE: If you happen to stumble across a large (> 8 digits) prime number, the cell could take a very, very long time
# to run and will likely crash your kernel. Just click kernel>interrupt if it looks like it's caught.

possible_prime = 9

for i in range(2, possible_prime):
    if (is_multiple(possible_prime, i)):
        print(possible_prime, 'is not prime')
        break
    if (i >= possible_prime/2):
        print(possible_prime, 'is prime')
        break
```

```
9 is not prime
```

Question 4: Writing Functions

In the following cell, complete a function that will take as its parameters a list and two integers x and y, iterate through the list, and replace any number in the list that is a multiple of x with y.

> Hint: use the is_multiple() function to streamline your code.

In [0]:
```
def replace_with y(lst, x, y):
    for i in range(len(lst)):
        if(is_multiple(lst[i],x)):
            lst[i] = y
    return lst
```

Part 2: Pandas Dataframes

We will be using Pandas dataframes for much of this class to organize and sort through economic data. Pandas (http://pandas.pydata.org/pandas-docs/stable/) is one of the most widely used Python libraries in data science. It is commonly used for data cleaning, and with good reason: it's very powerful and flexible, among many other things. Like we did with `numpy`, we will have to import `pandas`.

In [47]:
```
import pandas as pd
```

Creating dataframes

The rows and columns of a pandas dataframe are essentially a collection of lists stacked on top/next to each other. For example, if I wanted to store the top 10 movies and their ratings in a datatable, I could create 10 lists that each contain a rating and a corresponding title, and these lists would be the rows of the table:

In [48]:
```
top_10_movies = pd.DataFrame(data=np.array(
            [[9.2, 'The Shawshank Redemption (1994)'],
            [9.2, 'The Godfather (1972)'],
            [9., 'The Godfather: Part II (1974)'],
            [8.9, 'Pulp Fiction (1994)'],
            [8.9, "Schindler's List (1993)"],
            [8.9, 'The Lord of the Rings: The Return of the King (2003)'],
            [8.9, '12 Angry Men (1957)'],
            [8.9, 'The Dark Knight (2008)'],
            [8.9, 'Il buono, il brutto, il cattivo (1966)'],
            [8.8, 'The Lord of the Rings: The Fellowship of the Ring (2001)'
]]), columns=["Rating", "Movie"])
top_10_movies
```

Out[48]:

	Rating	Movie
0	9.2	The Shawshank Redemption (1994)
1	9.2	The Godfather (1972)
2	9.0	The Godfather: Part II (1974)
3	8.9	Pulp Fiction (1994)
4	8.9	Schindler's List (1993)
5	8.9	The Lord of the Rings: The Return of the King ...
6	8.9	12 Angry Men (1957)
7	8.9	The Dark Knight (2008)
8	8.9	Il buono, il brutto, il cattivo (1966)
9	8.8	The Lord of the Rings: The Fellowship of the R...

Alternatively, we can store data in a dictionary instead of in lists. A dictionary keeps a mapping of keys to a set of values, and each key is unique. Using our top 10 movies example, we could create a dictionary that contains ratings a key, and movie titles as another key.

In [50]:
```
top_10_movies_dict = {"Rating" : [9.2, 9.2, 9., 8.9, 8.9, 8.9, 8.9, 8.9, 8.9,
8.8],
                      "Movie" : ['The Shawshank Redemption (1994)',
                                 'The Godfather (1972)',
                                 'The Godfather: Part II (1974)',
                                 'Pulp Fiction (1994)',
                                 "Schindler's List (1993)",
                                 'The Lord of the Rings: The Return of the King
(2003)',
                                 '12 Angry Men (1957)',
                                 'The Dark Knight (2008)',
                                 'Il buono, il brutto, il cattivo (1966)',
                                 'The Lord of the Rings: The Fellowship of the
 Ring (2001)']}
```

Now, we can use this dictionary to create a table with columns `Rating` and `Movie`

In [51]:
```
top_10_movies_2 = pd.DataFrame(data=top_10_movies_dict, columns=["Rating", "Mo
vie"])
top_10_movies_2
```

Out[51]:

	Rating	Movie
0	9.2	The Shawshank Redemption (1994)
1	9.2	The Godfather (1972)
2	9.0	The Godfather: Part II (1974)
3	8.9	Pulp Fiction (1994)
4	8.9	Schindler's List (1993)
5	8.9	The Lord of the Rings: The Return of the King ...
6	8.9	12 Angry Men (1957)
7	8.9	The Dark Knight (2008)
8	8.9	Il buono, il brutto, il cattivo (1966)
9	8.8	The Lord of the Rings: The Fellowship of the R...

Notice how both ways return the same table! However, the list method created the table by essentially taking the lists and making up the rows of the table, while the dictionary method took the keys from the dictionary to make up the columns of the table. In this way, dataframes can be viewed as a collection of basic data structures, either through collecting rows or columns.

Reading in Dataframes

Luckily for you, most datatables in this course will be premade and given to you in a form that is easily read into a pandas method, which creates the table for you. A common file type that is used for economic data is a Comma-Separated Values (.csv) file, which stores tabular data. It is not necessary for you to know exactly how .csv files store data, but you should know how to read a file in as a pandas dataframe. You can use the "read_csv" method from pandas, which takes in one parameter which is the path to the csv file you are reading in.

We will read in a .csv file that contains quarterly real GDI, real GDP, and nominal GDP data in the U.S. from 1947 to the present.

In [55]:
```
import pandas as pd

# Run this cell to read in the table
accounts = pd.read_csv("Quarterly_Accounts.csv")
```

The `pd.read_csv` function expects a path to a .csv file as its input, and will return a data table created from the data contained in the csv. We have provided `Quarterly_Accouunts.csv` in the data directory, which is all contained in the current working directory (aka the folder this assignment is contained in). For this reason, we must specify to the `read_csv` function that it should look for the csv in the data directory, and the `/` indicates that `Quarterly_Accounts.csv` can be found there.

Here is a sample of some of the rows in this datatable:

In [56]:
```
accounts.head()
```

Out[56]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
0	1947	Q1	1912.5	1934.5	243.1
1	1947	Q2	1910.9	1932.3	246.3
2	1947	Q3	1914.0	1930.3	250.1
3	1947	Q4	1932.0	1960.7	260.3
4	1948	Q1	1984.4	1989.5	266.2

Indexing Dataframes

Oftentimes, tables will contain a lot of extraneous data that muddles our data tables, making it more difficult to quickly and accurately obtain the data we need. To correct for this, we can select out columns or rows that we need by indexing our dataframes.

The easiest way to index into a table is with square bracket notation. Suppose you wanted to obtain all of the Real GDP data from the data. Using a single pair of square brackets, you could index the table for `"Real GDP"`

In [57]:
```
# Run this cell and see what it outputs
accounts["Real GDP"]
```

Out[57]:

0	1934.5
1	1932.3
2	1930.3
3	1960.7
4	1989.5
5	2021.9
6	2033.2
7	2035.3
8	2007.5
9	2000.8
10	2022.8
11	2004.7
12	2084.6
13	2147.6
14	2230.4
15	2273.4
16	2304.5
17	2344.5
18	2392.8
19	2398.1
20	2423.5
21	2428.5
22	2446.1
23	2526.4
24	2573.4
25	2593.5
26	2578.9
27	2539.8
28	2528.0
29	2530.7
...	...
251	14541.9
252	14604.8
253	14745.9
254	14845.5
255	14939.0
256	14881.3
257	14989.6
258	15021.1
259	15190.3
260	15291.0
261	15362.4
262	15380.8
263	15384.3
264	15491.9
265	15521.6
266	15641.3
267	15793.9
268	15757.6
269	15935.8
270	16139.5
271	16220.2
272	16350.0
273	16460.9
274	16527.6
275	16547.6
276	16571.6

```
277    16663.5
278    16778.1
279    16851.4
280    16903.2
Name: Real GDP, Length: 281, dtype: float64
```

Notice how the above cell returns an array of all the real GDP values in their original order. Now, if you wanted to get the first real GDP value from this array, you could index it with another pair of square brackets:

In [58]:
```
accounts["Real GDP"][0]
```

Out[58]:
```
1934.5
```

Pandas columns have many of the same properties as numpy arrays. Keep in mind that pandas dataframes, as well as many other data structures, are zero-indexed, meaning indexes start at 0 and end at the number of elements minus one.

If you wanted to create a new datatable with select columns from the original table, you can index with double brackets.

In [63]:
```
## Note: .head() returns the first five rows of the table
accounts[["Year", "Quarter", "Real GDP", "Real GDI"]].head()
```

Out[63]:

	Year	Quarter	Real GDP	Real GDI
0	1947	Q1	1934.5	1912.5
1	1947	Q2	1932.3	1910.9
2	1947	Q3	1930.3	1914.0
3	1947	Q4	1960.7	1932.0
4	1948	Q1	1989.5	1984.4

You can also use column indices instead of names.

In [68]:
```
accounts.iloc[[0, 1, 2, 3]].head()
```

Out[68]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
0	1947	Q1	1912.5	1934.5	243.1
1	1947	Q2	1910.9	1932.3	246.3
2	1947	Q3	1914.0	1930.3	250.1
3	1947	Q4	1932.0	1960.7	260.3

Alternatively, you can also get rid of columns you dont need using `.drop()`

In [69]:
```
accounts.drop("Nominal GDP", axis=1).head()
```

Out[69]:

	Year	Quarter	Real GDI	Real GDP
0	1947	Q1	1912.5	1934.5
1	1947	Q2	1910.9	1932.3
2	1947	Q3	1914.0	1930.3
3	1947	Q4	1932.0	1960.7
4	1948	Q1	1984.4	1989.5

Finally, you can use square bracket notation to index rows by their indices with a single set of brackets. You must specify a range of values for which you want to index. For example, if I wanted the 20th to 30th rows of `accounts`:

In [70]:
```
accounts[20:31]
```

Out[70]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
20	1952	Q1	2398.3	2423.5	360.2
21	1952	Q2	2412.6	2428.5	361.4
22	1952	Q3	2435.0	2446.1	368.1
23	1952	Q4	2509.5	2526.4	381.2
24	1953	Q1	2554.3	2573.4	388.5
25	1953	Q2	2572.2	2593.5	392.3
26	1953	Q3	2555.7	2578.9	391.7
27	1953	Q4	2504.1	2539.8	386.5
28	1954	Q1	2510.1	2528.0	385.9
29	1954	Q2	2514.5	2530.7	386.7
30	1954	Q3	2537.1	2559.4	391.6

Filtering Data

As you can tell from the previous, indexing rows based on indices is only useful when you know the specific set of rows that you need, and you can only really get a range of entries. Working with data often involves huge datasets, making it inefficient and sometimes impossible to know exactly what indices to be looking at. On top of that, most data analysis concerns itself with looking for patterns or specific conditions in the data, which is impossible to look for with simple index based sorting.

Thankfully, you can also use square bracket notation to filter out data based on a condition. Suppose we only wanted real GDP and nominal GDP data from the 21st century:

In [71]:
```
accounts[accounts["Year"] >= 2000][["Real GDP", "Nominal GDP"]]
```

Out[71]:

	Real GDP	Nominal GDP
212	12359.1	10031.0
213	12592.5	10278.3
214	12607.7	10357.4
215	12679.3	10472.3
216	12643.3	10508.1
217	12710.3	10638.4
218	12670.1	10639.5
219	12705.3	10701.3
220	12822.3	10834.4
221	12893.0	10934.8
222	12955.8	11037.1
223	12964.0	11103.8
224	13031.2	11230.1
225	13152.1	11370.7
226	13372.4	11625.1
227	13528.7	11816.8
228	13606.5	11988.4
229	13706.2	12181.4
230	13830.8	12367.7
231	13950.4	12562.2
232	14099.1	12813.7
233	14172.7	12974.1
234	14291.8	13205.4
235	14373.4	13381.6
236	14546.1	13648.9
237	14589.6	13799.8
238	14602.6	13908.5
239	14716.9	14066.4
240	14726.0	14233.2
241	14838.7	14422.3
...	...	...
251	14541.9	14566.5
252	14604.8	14681.1
253	14745.9	14888.6
254	14845.5	15057.7

	Real GDP	Nominal GDP
255	14939.0	15230.2
256	14881.3	15238.4
257	14989.6	15460.9
258	15021.1	15587.1
259	15190.3	15785.3
260	15291.0	15973.9
261	15362.4	16121.9
262	15380.8	16227.9
263	15384.3	16297.3
264	15491.9	16475.4
265	15521.6	16541.4
266	15641.3	16749.3
267	15793.9	16999.9
268	15757.6	17031.3
269	15935.8	17320.9
270	16139.5	17622.3
271	16220.2	17735.9
272	16350.0	17874.7
273	16460.9	18093.2
274	16527.6	18227.7
275	16547.6	18287.2
276	16571.6	18325.2
277	16663.5	18538.0
278	16778.1	18729.1
279	16851.4	18905.5
280	16903.2	19057.7

69 rows × 2 columns

The `accounts` table is being indexed by the condition `accounts["Year"] >= 2000`, which returns a table where only rows that have a "Year" greater than 2000 is returned. We then index this table with the double bracket notation from the previous section to only get the real GDP and nominal GDP columns.

Suppose now we wanted a table with data from the first quarter, and where the real GDP was less than 5000 or nominal GDP is greater than 15,000.

In [72]:
```
accounts[(accounts["Quarter"] == "Q1") & ((accounts["Real GDP"] < 5000) | (acc
ounts["Nominal GDP"] > 15000))]
```

Out[72]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
0	1947	Q1	1912.5	1934.5	243.1
4	1948	Q1	1984.4	1989.5	266.2
8	1949	Q1	2001.5	2007.5	275.4
12	1950	Q1	2060.1	2084.6	281.2
16	1951	Q1	2281.0	2304.5	336.4
20	1952	Q1	2398.3	2423.5	360.2
24	1953	Q1	2554.3	2573.4	388.5
28	1954	Q1	2510.1	2528.0	385.9
32	1955	Q1	2661.6	2683.8	413.8
36	1956	Q1	2775.4	2770.0	440.5
40	1957	Q1	2862.0	2854.5	470.6
44	1958	Q1	2779.9	2772.7	468.4
48	1959	Q1	2976.5	2976.6	511.1
52	1960	Q1	3121.9	3123.2	543.3
56	1961	Q1	3109.9	3102.3	545.9
60	1962	Q1	3328.6	3336.8	595.2
64	1963	Q1	3469.1	3456.1	622.7
68	1964	Q1	3658.6	3672.7	671.1
72	1965	Q1	3885.5	3873.5	719.2
76	1966	Q1	4167.8	4201.9	797.3
80	1967	Q1	4286.5	4324.9	846.0
84	1968	Q1	4465.6	4490.6	911.1
88	1969	Q1	4665.4	4691.6	995.4
92	1970	Q1	4690.4	4707.1	1053.5
96	1971	Q1	4778.0	4834.3	1137.8
256	2011	Q1	14924.4	14881.3	15238.4
260	2012	Q1	15500.4	15291.0	15973.9
264	2013	Q1	15642.7	15491.9	16475.4
268	2014	Q1	15912.8	15757.6	17031.3
272	2015	Q1	16599.6	16350.0	17874.7
276	2016	Q1	16776.1	16571.6	18325.2
280	2017	Q1	16992.1	16903.2	19057.7

Many different conditions can be included to filter, and you can use `&` and `|` operators to connect them together. Make sure to include parantheses for each condition!

Another way to reorganize data to make it more convenient is to sort the data by the values in a specific column. For example, if we wanted to find the highest real GDP since 1947, we could sort the table for real GDP:

In [73]:
```
accounts.sort_values("Real GDP")
```

Out[73]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
2	1947	Q3	1914.0	1930.3	250.1
1	1947	Q2	1910.9	1932.3	246.3
0	1947	Q1	1912.5	1934.5	243.1
3	1947	Q4	1932.0	1960.7	260.3
4	1948	Q1	1984.4	1989.5	266.2
9	1949	Q2	1995.9	2000.8	271.7
11	1949	Q4	1979.6	2004.7	271.0
8	1949	Q1	2001.5	2007.5	275.4
5	1948	Q2	2030.2	2021.9	272.9
10	1949	Q3	2007.9	2022.8	273.3
6	1948	Q3	2031.5	2033.2	279.5
7	1948	Q4	2041.6	2035.3	280.7
12	1950	Q1	2060.1	2084.6	281.2
13	1950	Q2	2144.4	2147.6	290.7
14	1950	Q3	2225.9	2230.4	308.5
15	1950	Q4	2268.9	2273.4	320.3
16	1951	Q1	2281.0	2304.5	336.4
17	1951	Q2	2321.3	2344.5	344.5
18	1951	Q3	2362.0	2392.8	351.8
19	1951	Q4	2382.7	2398.1	356.6
20	1952	Q1	2398.3	2423.5	360.2
21	1952	Q2	2412.6	2428.5	361.4
22	1952	Q3	2435.0	2446.1	368.1
23	1952	Q4	2509.5	2526.4	381.2
28	1954	Q1	2510.1	2528.0	385.9
29	1954	Q2	2514.5	2530.7	386.7
27	1953	Q4	2504.1	2539.8	386.5
30	1954	Q3	2537.1	2559.4	391.6
24	1953	Q1	2554.3	2573.4	388.5
26	1953	Q3	2555.7	2578.9	391.7
...	...	...	...	...	...
244	2008	Q1	14842.2	14889.5	14668.4
246	2008	Q3	14767.0	14891.6	14843.0
242	2007	Q3	14822.4	14938.5	14569.7
255	2010	Q4	14904.9	14939.0	15230.2

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
245	2008	Q2	14832.4	14963.4	14813.0
257	2011	Q2	14996.1	14989.6	15460.9
243	2007	Q4	14816.6	14991.8	14685.3
258	2011	Q3	15093.1	15021.1	15587.1
259	2011	Q4	15217.0	15190.3	15785.3
260	2012	Q1	15500.4	15291.0	15973.9
261	2012	Q2	15522.8	15362.4	16121.9
262	2012	Q3	15517.1	15380.8	16227.9
263	2012	Q4	15650.6	15384.3	16297.3
264	2013	Q1	15642.7	15491.9	16475.4
265	2013	Q2	15719.8	15521.6	16541.4
266	2013	Q3	15752.0	15641.3	16749.3
268	2014	Q1	15912.8	15757.6	17031.3
267	2013	Q4	15851.3	15793.9	16999.9
269	2014	Q2	16136.1	15935.8	17320.9
270	2014	Q3	16327.9	16139.5	17622.3
271	2014	Q4	16520.8	16220.2	17735.9
272	2015	Q1	16599.6	16350.0	17874.7
273	2015	Q2	16700.6	16460.9	18093.2
274	2015	Q3	16726.7	16527.6	18227.7
275	2015	Q4	16789.8	16547.6	18287.2
276	2016	Q1	16776.1	16571.6	18325.2
277	2016	Q2	16783.0	16663.5	18538.0
278	2016	Q3	16953.0	16778.1	18729.1
279	2016	Q4	16882.1	16851.4	18905.5
280	2017	Q1	16992.1	16903.2	19057.7

281 rows × 5 columns

But wait! The table looks like it's sorted in increasing order. This is because `sort_values` defaults to ordering the column in ascending order. To correct this, add in the extra optional parameter

In [74]:
```
accounts.sort_values("Real GDP", ascending=False)
```

Out[74]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
280	2017	Q1	16992.1	16903.2	19057.7
279	2016	Q4	16882.1	16851.4	18905.5
278	2016	Q3	16953.0	16778.1	18729.1
277	2016	Q2	16783.0	16663.5	18538.0
276	2016	Q1	16776.1	16571.6	18325.2
275	2015	Q4	16789.8	16547.6	18287.2
274	2015	Q3	16726.7	16527.6	18227.7
273	2015	Q2	16700.6	16460.9	18093.2
272	2015	Q1	16599.6	16350.0	17874.7
271	2014	Q4	16520.8	16220.2	17735.9
270	2014	Q3	16327.9	16139.5	17622.3
269	2014	Q2	16136.1	15935.8	17320.9
267	2013	Q4	15851.3	15793.9	16999.9
268	2014	Q1	15912.8	15757.6	17031.3
266	2013	Q3	15752.0	15641.3	16749.3
265	2013	Q2	15719.8	15521.6	16541.4
264	2013	Q1	15642.7	15491.9	16475.4
263	2012	Q4	15650.6	15384.3	16297.3
262	2012	Q3	15517.1	15380.8	16227.9
261	2012	Q2	15522.8	15362.4	16121.9
260	2012	Q1	15500.4	15291.0	15973.9
259	2011	Q4	15217.0	15190.3	15785.3
258	2011	Q3	15093.1	15021.1	15587.1
243	2007	Q4	14816.6	14991.8	14685.3
257	2011	Q2	14996.1	14989.6	15460.9
245	2008	Q2	14832.4	14963.4	14813.0
255	2010	Q4	14904.9	14939.0	15230.2
242	2007	Q3	14822.4	14938.5	14569.7
246	2008	Q3	14767.0	14891.6	14843.0
244	2008	Q1	14842.2	14889.5	14668.4
...	...	...	...	...	...
26	1953	Q3	2555.7	2578.9	391.7
24	1953	Q1	2554.3	2573.4	388.5
30	1954	Q3	2537.1	2559.4	391.6
27	1953	Q4	2504.1	2539.8	386.5

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
29	1954	Q2	2514.5	2530.7	386.7
28	1954	Q1	2510.1	2528.0	385.9
23	1952	Q4	2509.5	2526.4	381.2
22	1952	Q3	2435.0	2446.1	368.1
21	1952	Q2	2412.6	2428.5	361.4
20	1952	Q1	2398.3	2423.5	360.2
19	1951	Q4	2382.7	2398.1	356.6
18	1951	Q3	2362.0	2392.8	351.8
17	1951	Q2	2321.3	2344.5	344.5
16	1951	Q1	2281.0	2304.5	336.4
15	1950	Q4	2268.9	2273.4	320.3
14	1950	Q3	2225.9	2230.4	308.5
13	1950	Q2	2144.4	2147.6	290.7
12	1950	Q1	2060.1	2084.6	281.2
7	1948	Q4	2041.6	2035.3	280.7
6	1948	Q3	2031.5	2033.2	279.5
10	1949	Q3	2007.9	2022.8	273.3
5	1948	Q2	2030.2	2021.9	272.9
8	1949	Q1	2001.5	2007.5	275.4
11	1949	Q4	1979.6	2004.7	271.0
9	1949	Q2	1995.9	2000.8	271.7
4	1948	Q1	1984.4	1989.5	266.2
3	1947	Q4	1932.0	1960.7	260.3
0	1947	Q1	1912.5	1934.5	243.1
1	1947	Q2	1910.9	1932.3	246.3
2	1947	Q3	1914.0	1930.3	250.1

281 rows × 5 columns

Now we can clearly see that the highest real GDP was attained in the first quarter of this year, and had a value of 16903.2

Useful Functions for Numeric Data

Here are a few useful functions when dealing with numeric data columns. To find the minimum value in a column, call `min()` on a column of the table.

In [75]:
```
accounts["Real GDP"].min()
```

Out[75]:
```
1930.3
```

To find the maximum value, call `max()` .

In [76]:
```
accounts["Nominal GDP"].max()
```

Out[76]:
```
19057.7
```

And to find the average value of a column, use `mean()` .

In [81]:
```
accounts["Real GDI"].mean()
```

Out[81]:
```
7890.370462633456
```

Part 3: Visualization

Now that you can read in data and manipulate it, you are now ready to learn about how to visualize data. To begin, run the cells below to import the required packages we will be using.

In [82]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

We will be using US unemployment data from FRED (https://fred.stlouisfed.org/) to show what we can do with data. The statement below will put the csv file into a pandas DataFrame.

In [83]:
```
import pandas as pd

unemployment_data = pd.read_csv("detailed_unemployment.csv")
unemployment_data.head()
```

Out[83]:

	date	total_unemployed	more_than_15_weeks	not_in_labor_searched_for_work	multi_jobs
0	11/1/10	16.9	8696	2531	6708
1	12/1/10	16.6	8549	2609	6899
2	1/1/11	16.2	8393	2800	6816
3	2/1/11	16.0	8175	2730	6741
4	3/1/11	15.9	8166	2434	6735

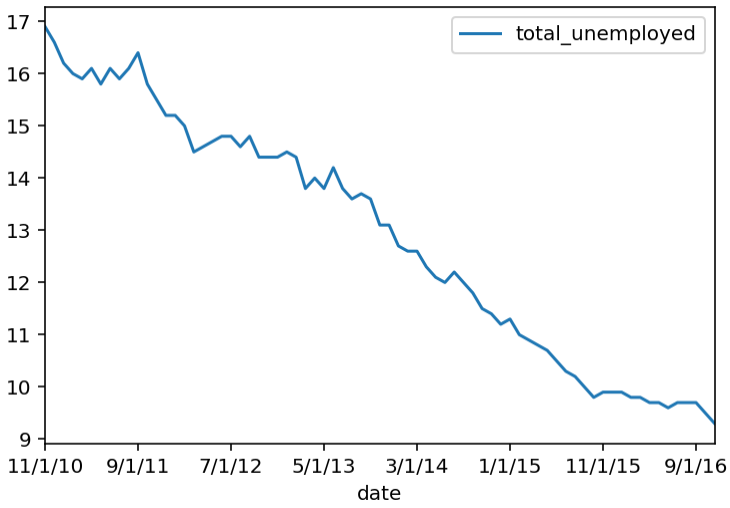
One of the advantages of pandas is its built-in plotting methods. We can simply call `.plot()` on a dataframe to plot columns against one another. All that we have to do is specify which column to plot on which axis. Something special that pandas does is attempt to automatically parse dates into something that it can understand and order them sequentially.

Sidenote: `total_unemployed` is a percent.

In [84]:
```
unemployment_data.plot(x='date', y='total_unemployed')
```

Out[84]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9fdc5d8358>
```

Out[84]:



The base package for most plotting in Python is `matplotlib`. Below we will look at how to plot with it. First we will extract the columns that we are interested in, then plot them in a scatter plot. Note that `plt` is the common convention for `matplotlib.pyplot`.

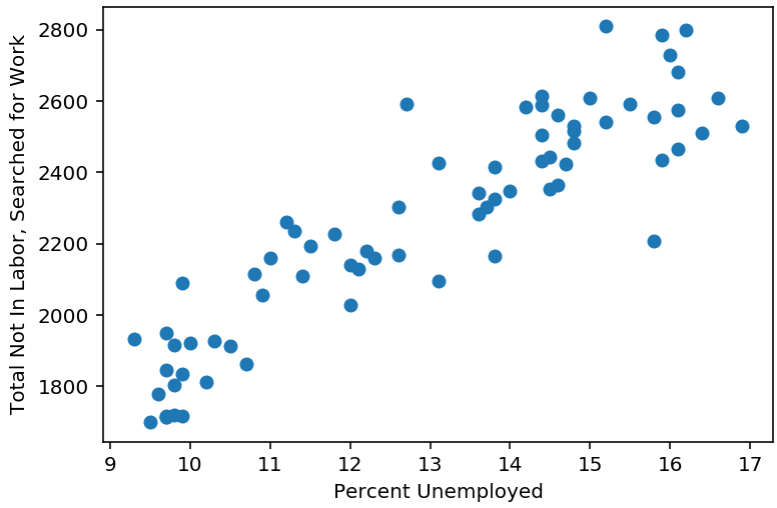
In [87]:
```
total_unemployed = unemployment_data['total_unemployed']
not_labor = unemployment_data['not_in_labor_searched_for_work']

#Plot the data by inputting the x and y axis
plt.scatter(total_unemployed, not_labor)

# we can then go on to customize the plot with labels
plt.xlabel("Percent Unemployed")
plt.ylabel("Total Not In Labor, Searched for Work")
```

Out[87]:
```
Text(0,0.5,'Total Not In Labor, Searched for Work')
```

Out[87]:



Though matplotlib is sometimes considered an "ugly" plotting tool, it is powerful. It is highly customizable and is the foundation for most Python plotting libraries. Check out the documentation (https://matplotlib.org/api/pyplot_summary.html) to get a sense of all of the things you can do with it, which extend far beyond scatter and line plots. An arguably more attractive package is seaborn (https://seaborn.pydata.org/), which we will go over in future notebooks.

Question 5: Plotting

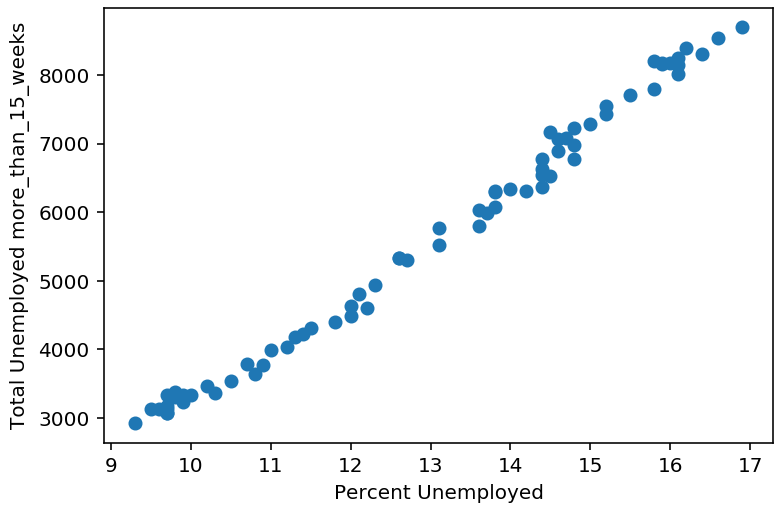
Try plotting the total percent of people unemployed vs those unemployed for more than 15 weeks.

In [89]:
```
total_unemployed = unemployment_data['total_unemployed']
unemp_15_weeks = unemployment_data['more_than_15_weeks']

plt.scatter(total_unemployed, unemp_15_weeks)
plt.xlabel("Percent Unemployed")
plt.ylabel("Total Unemployed more_than_15_weeks")

# note: plt.show() is the equivalent of print, but for graphs
plt.show()
```

Out[89]:



Math Concepts Review

These questions are math review.

```
For questions that are to be answered numerically, there is a code cell that starts
with __# ANSWER__ and has a variable currently set to underscores. Replace those un
derscores with your final answer. It is okay to make other computations in that cel
l and others, so long a the given variable matches your answer.
```

For free response questions, write your answers in the provided markdown cell that starts with ANSWER:. Do not change the heading, and write your entire answer in that one cell.

Math as a Tool

Suppose a quantity grows at a steady proportional rate of 3% per year...

...How long will it take to double?

In [0]:
```
# ANSWER
TIME_TO_DOUBLE =24 __
```

Quadruple?

In [0]:
```
# ANSWER
TIME_TO_QUADRUPLE = 48__
```

Grow 1024-fold?

In [0]:
```
# ANSWER
TIME_TO_1024 = 240__
```

Suppose we have a quantity x(t) that varies over time following the equation:

$\frac{dx(t)}{dt} = -(0.06)x + 0.36$

$\frac{dx(t)}{dt} - \frac{d(6)}{dt} = -(0.06)(x - 6)$

$\frac{dx(t)}{dt} = -(0.06)(x - 6)$

$\frac{dy(t)}{dt} = -(0.06)y$

Without integrating the equation:

1. Tell me what the long-run steady-state value of x--that is, the limit of x as t approaches in infinity--is going to be.

In [0]:
```
steady_state_val = 6
```

2. Suppose that the value of x at time $t = 0$, $x(0)$ equals 12. Once again, without integrating the equation, tell me how long it will take x to close half the distance between its initial value of 12 and its steady-state value.

In [0]:
```
half_dist_time = 12
```

3. How long will it take to close 3/4 of the distance?

In [0]:
```
three_fourth_time = 24
```

4. $7/8$ of the distance?

In [0]:
```
seven_eighth_time = 36
```

5. $15/16$ of the distance?

```
In [1]: fifteen_sixteenth = 48
```

Now you are allowed to integrate $\frac{dx(t)}{dt} = -(0.06)x + 0.36$.

1. Write down and solve the indefinite integral.

ANSWER:
$x = (x_0 - 6)e^{-0.06t}$

2. Write down and solve the definite integral for the initial condition $x(0) = 12$.

ANSWER: $x = 6e^{-0.06t}$

3. Write down and solve the definite integral for the initial condition $x(0) = 6$.

ANSWER:
$x = 0$

Suppose we have a quantity $z = (\frac{x}{y})^{\beta}$

Suppose x is growing at 4% per year and that $\beta = 1/4$:

1. How fast is z growing if y is growing at 0% per year?

```
In [0]: zero_per_growth = 0.99%
```

2. If y is growing at 2% per year?

```
In [0]: two_per_growth = 0.487%
```

3. If y is growing at 4% per year?

In [0]:
```
four_per_growth = 0%
```

Rule of 72 (Use it for the next four questions)

1. If a quantity grows at about 3% per year, how long will it take to double?

In [0]:
```
time_to_double = 24__
```

2. If a quantity shrinks at about 4% per year, how long will it take it to halve itself?

In [0]:
```
time_to_half =18 __
```

3. If a quantity doubles five times, how large is it relative to its original value?

In [0]:
```
doubled_five_times_ratio = _32_
```

4. If a quantity halves itself three times, how large is it relative to its original value?

In [0]:
```
halved_three_times_ratio =1/8__
```

Interactive Model for Rule of 72

In future problem sets, you will build models of your own, but for now, look over this code. Its a simple model that shows what happens as you adjust a single parameter (the interest rate) and its effect on the outcome (the time to double). First we need to make sure all of our packages are imported.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
from ipywidgets import interact, IntSlider
%matplotlib inline
```

Our model is going to be graph that shows what happens as the interest rate varies.

In [5]:
```
def graph_rule_of_72(interest_rate):
    # np.linspace takes values evenly spaced between a stop and end point. In
 this case,
    # will take 30 values between 1 and 10. These will be our x values in the
 graph.
    x = np.linspace(1,10,30)

    # Here we create are corresponding y values
    y = 72 / x

    print('Time to double:', 72 / interest_rate, 'years')

    # graphing our lines
    plt.plot(x,y)
    # graphing the specific point for our interest_rate
    plt.scatter(interest_rate, 72 / interest_rate, c='r')

    plt.xlabel('interest rate (%)')
    plt.ylabel('time (years)')
    plt.show()
```

When we call `interact`, select the function that we want to interact with (`graph_rule_of_72`) and tell it what the value we want its parameters to take on. In this case, `graph_rule_of_72` only takes one parameter, `interest_rate`, and we choose to put an adjustable slider there. You can check out the ipywidget examples (https://github.com/jupyter-widgets/ipywidgets/blob/master/docs/source/examples/Index.ipynb) for more uses.

In [4]:
```
interact(graph_rule_of_72, interest_rate=IntSlider(min=1,max=10,step=1))
```

Out[4]:

Out[4]: <function __main__.graph_rule_of_72>

Why do DeLong and Olney think that the interest rate and the level of the stock market are important macroeconomic variables?

ANSWER:interest rate and the level of the stock market are closely related.These two variables will effect consumers' will to purchase goods,investors' will to borrow monoey to invest,etc.which may affect many other macroeconomic variables.

What are the principal flaws in using national product per worker as a measure of material welfare? Given these flaws, why do we use it anyway?

ANSWER:It does not value the quality of the environment,workers' leisure time,effective distribution of income,etc.But having a large national product per worker does help government to afford better environment,health care,education anyway,many indicators of quality of life are positively correlated with GDP.Also,it only calculates the value of final goods instead of intermidiate goods,thus there are no double count.

What is the difference between the nominal interest rate and the real interest rate? Why do DeLong and Olney think that the real interest rate is more important?

ANSWER:nominal interest rate shows the dollar value of a deposit,while real interest rate shows the purchasing power of a deposit/debit,real interst rate is corrected for inflation.Because there exists inflation,so real interest rate can better shows the purchasing power of deposit.

Review: Measuring the Economy Concepts and Quantities

National Income and Product Accounting

Explain whether or not, why, and how the following items are included in the calculations of national product:

1. Increases in business inventories.

ANSWER:included,this activity can be considered as increase in investment.

2. Fees earned by real estate agents on selling existing homes.

ANSWER:included.this is part of wages.

3. Social Security checks written by the government.

ANSWER:not included.Social security checks is transfer payment,which does not involve purchase of goods and service.

4. Building of a new dam by the Army Corps of Engineers.

ANSWER:included.this can be considered as increase in government purchase.

5. Interest that your parents pay on the mortgage they have on their house.

ANSWER:included.this is part of interest income,which should be counted in to national product.

6. Purchases of foreign-made trucks by American residents

ANSWER:included.consumption increase,but net exports decrease,so national product does not change.

In or Out of National Product? And Why

Explain whether or not, why, and how the following items are included in the calculation of national product:

1. The sale for $25,000 of an automobile that cost $20,000 to manufacture that had been produced here at home last year and carried over in inventory.

ANSWER:included.inventory investment decreases,but consumption increases by 25000 dollars,national product does not change.

2. The sale for $35,000 of an automobile that cost $25,000 to manufacture newly- made at home this year.

ANSWER:included.This can be considered as increase in consumption.

3. The sale for $45,000 of an automobile that cost $30,000 to manufacture that was newly-made abroad this year and imported.

ANSWER:included.Consumption increases by 45000 dollars,while net export falls down.

4. The sale for $25,000 of an automobile that cost $20,000 to manufacture that was made abroad and imported last year.

ANSWER:not included.It doesn't involve purchase of goods and services this year,so it can't be calculated into national product this year.

In or Out of National Product? And Why II

Explain whether or not, why, and how the following items are included in the calculation of GDP:

1. The purchase for $500 of a dishwasher produced here at home this year.

ANSWER:included.consumption increases by $500.

2. The purchase for $500 of a dishwasher made abroad this year.

ANSWER:included.consumption increases by $500 while net export falls down by $500.

3. The purchase for $500 of a used dishwasher.

ANSWER:not included.This transaction is just an allocation of values in the past.There are no new products and services produced this year.

4. The manufacture of a new dishwasher here at home for $500 of a dishwasher that then nobody wants to buy.

ANSWER:included.inventory investment rises by $500.

Components of National Income and Product

Suppose that the appliance store buys a refrigerator from the manufacturer on December 15, 2018 for $600, and that you then buy that refrigerator on January 15, 2019 for $1000:

1. What is the contribution to GDP in 2018?

```
In [0]: contribution_2018 = $600
```

2. How is the refrigerator accounted for in the NIPA in 2019?

ANSWER:

3. What is the contribution to GDP in 2019?

```
In [2]: contribution_2019 = ___
```

4. How is the refrigerator accounted for in the NIPA in 2019?

ANSWER:

In [9]:
```
# These lines are reading in CSV files and creating dataframes from then, you
 don't have to change about them!

import pandas as pd
import numpy as np

unemployment = pd.read_csv("detailed_unemployment.csv")
quarterly_acc = pd.read_csv("Quarterly_Accounts.csv")
from_2007 = quarterly_acc.loc[(quarterly_acc["Year"].isin(np.arange(2007, 2018
)))]
```

Estimating National Product

The Bureau of Economic Analysis measures national product in two different ways: as total expenditure on the economy's output of goods and services and as the total income of everyone in the economy. Since – as you learned in earlier courses – these two things are the same, the two approaches should give the same answer. But in practice they do not.

We have provided a data table `quarterly_gdp` that contains quarterly data on real GDP measured on the expenditure side (referred to in the National Income and Product Accounts as "Real Gross Domestic Product, chained dollars") and real GDP measured on the income side (referred to as "Real Gross Domestic Income, chained dollars"). The table refers to Real Gross Dometic Product as "Real GDP" and to Real Gross Dometic Income as "Real GDI", and they are measured in billions of dollars. (Note: You will not have to use Nominal GDP)

Another table, `from_2007`, has been created from `quarterly_gdp`, and includes information from 2007 to 2017. Below is a snippet from `from_2007`:

In [10]:
```
from_2007.head(10)
```

Out[10]:

	Year	Quarter	Real GDI	Real GDP	Nominal GDP
240	2007	Q1	14882.6	14726.0	14233.2
241	2007	Q2	14904.1	14838.7	14422.3
242	2007	Q3	14822.4	14938.5	14569.7
243	2007	Q4	14816.6	14991.8	14685.3
244	2008	Q1	14842.2	14889.5	14668.4
245	2008	Q2	14832.4	14963.4	14813.0
246	2008	Q3	14767.0	14891.6	14843.0
247	2008	Q4	14479.5	14577.0	14549.9
248	2009	Q1	14257.8	14375.0	14383.9
249	2009	Q2	14259.2	14355.6	14340.4

1. Compute the growth rate at an annual rate of each of the two series by quarter for 2007:Q1–2012:Q4.

In [0]:
```
gdi_rate = [0.780%, 0.727%, 0.611%, 0.537%]
gdp_rate = [0.855%, 0.854%, 0.948%, 1.120%]
```

2. Describe any two things you see when you compare the two series that you find interesting, and explain why you find them interesting.

ANSWER:the curve for gdp_rate and gdi_rate doesn't follow the same path,but they have the tend to move together over time.This is interesting because in theory, the two numbers should be the same,as both are designed to measure the aggregate growth of the economy,only in different methods.But actually,they give out different data.There may be some meausuremen

Calculating Real Magnitudes:

1. When you calculate real national product, do you do so by dividing nominal national product by the price level or by subtracting the price level from nominal national product?

ANSWER:by dividing nominal national product by the price level.

2. When you calculate the real interest rate, do you do so by dividing the nominal interest rate by the price level or by subtracting the inflation rate from the nominal interest rate?

ANSWER:by subtracting the inflation rate from the nominal interest rate.

3. Are your answers to (a) and (b) the same? Why or why not?

ANSWER:not the same.for question(a),real national product = nominal national product/(gdp deflator/100),gdp deflator represents the price level. for(b),nominal interest rate=real interest rate+inflation rate,so inflation rate should be subtracted from nominal interest rate.

Unemployment Rate

Use the `unemployment` table provided to answer the following questions. ***All numbers (other than percents) are in the thousands.***

Here are the first five entries of the table.

In [11]:
```
unemployment.head()
```

Out[11]:

	date	total_unemployed	more_than_15_weeks	not_in_labor_searched_for_work	multi_jobs
0	11/1/10	16.9	8696	2531	6708
1	12/1/10	16.6	8549	2609	6899
2	1/1/11	16.2	8393	2800	6816
3	2/1/11	16.0	8175	2730	6741
4	3/1/11	15.9	8166	2434	6735

What, roughly, was the highest level the U.S. unemployment rate (measured as Percent Unemployed of Labor Force in the table) reached in:

1. The 20th century?

In [0]:
```
unemployment_20th = _24.9%__
```

2. The past fifty years?

In [0]:
```
unemployment_past_50 = _10.8%__
```

3. The twenty years before 2006?

In [0]:
```
unemployment_before_2006 = 7.4%___
```

4. Given your answers to (1) through (3), Do you think there is a connection between your answer to the question above and the fact that Federal Reserve Chair Alan Greenspan received a five-minute standing ovation at the end of the first of many events marking his retirement in 2005?

ANSWER:yes.we can learn from the datatable that unemployment rate were kept down during the twenty years before 2006,so Alan Greenspan made remarkable achievements in lowing down the unemployment rate in his tenure.

The State of the Labor Market

1. About how many people lose or quit their jobs in an average year?

In [0]:
```
average_quitters = 78.84k
```

2. About how many people get jobs in an average year?

In [1]:
```
average_getters = 1368.09k
```

3. About how many people are unemployed in an average year?

In [0]:
```
average_unemployed = 6436.99k
```

4. About how many people are at work in an average year?

In [0]:
```
average_workers = 101425.9k
```

5. About how many people are unemployed now?

In [4]:
```
unemployed_now = 12433.6k
```

National Income Accounting:

1. What was the level of real GDP in 2005 dollars in 1970?

In [0]:
```
real_gdp_2005 = 76254 (1970 in 2005 dollars), 14103 (2005 in 1970 dollars)
```

2. What was the rate of inflation in the United States in 2000?

In [0]:
```
inflation_rate_2000 = 2.28%
```

3. Explain whether or not, how, and why the following items are included in the calculation of GDP: (i) rent you pay on an apartment, (ii) purchase of a used textbook, (iii) purchase of a new tank by the Department of Defense, (iv) watching an advertisement on youtube.

ANSWER:(i)included.GDP=W+R+I+PR+BT+D+F.this is part of rental income.(ii)not included.purchase of used textbooks involves allocation of value in the past rather than purchase of current goods & service.(iii)included.this transaction causes government purchase to rise.(iv)included.putting advertisements on youtube can be regarded as a form of investments to help sellers promote sales.

Congratulations, you have finished your first assignment for Econ 101B! Run the cell below to submit all of your work. Make sure to check on OK to make sure that it has uploaded.

Some materials this notebook were taken from Data 8 (http://data8.org/), CS 61A (http://cs61a.org/), and DS Modules (http://data.berkeley.edu/education/modules) lessons.