Lab 2: Data Types

Welcome to Lab 2!

Last time, we had our first look at Python and Jupyter notebooks. So far, we've only used Python to manipulate numbers. There's a lot more to life than numbers, so Python lets us represent many other types of data in programs.

In this lab, you'll first see how to represent and manipulate another fundamental type of data: text. A piece of text is called a *string* in Python.

You'll also see how to invoke *methods*. A method is very similar to a function. It just looks a little different because it's tied to a particular piece of data (like a piece of text or a number).

Last, you'll see how to work with datasets in Python -- *collections* of data, like the numbers 2 through 5 or the words "welcome", "to", and "lab".

1. Review: The building blocks of Python code

The two building blocks of Python code are *expressions* and *statements*. An **expression** is a piece of code that

- is self-contained, meaning it would make sense to write it on a line by itself, and
- usually has a value.

Here are two expressions that both evaluate to 3

```
3
5 - 2
```

One important form of an expression is the **call expression**, which first names a function and then describes its arguments. The function returns some value, based on its arguments. Some important mathematical functions are

Function	Description
`abs`	Returns the absolute value of its argument
`max`	Returns the maximum of all its arguments
`min`	Returns the minimum of all its arguments
`pow`	Raises its first argument to the power of its second argument
`round`	Round its argument to the nearest integer

Here are two call expressions that both evaluate to 3

```
abs(2 - 5)
max(round(2.8), min(pow(2, 10), -1 * pow(2, 10)))
```

A **statement** is a whole line of code. Some statements are just expressions. The expressions listed above are examples.

Other statements *make something happen* rather than *having a value*. After they are run, something in the world has changed. For example, an **assignment statement** assigns a value to a name.

A good way to think about this is that we're **evaluating the right-hand side** of the equals sign and **assigning it to the left-hand side**. Here are some assignment statements:

```
height = 1.3
the_number_five = abs(-5)
absolute_height_difference = abs(height - 1.688)
```

A key idea in programming is that large, interesting things can be built by combining many simple, uninteresting things. The key to understanding a complicated piece of code is breaking it down into its simple components.

For example, a lot is going on in the last statement above, but it's really just a combination of a few things. This picture describes what's going on.

Question 1. In the next cell, assign the name `new_year` to the larger number among the following two numbers:

1. the absolute value of $2^5 - 2^{11} - 2^1$, and
2. $5 \times 13 \times 31 + 2$.

Try to use just one statement (one line of code).

In [1]:
```
new_year = max((abs(2**5-2**11-2**1), (5*13*31+2)))
new_year
```

Out[1]:
```
2018
```

2. Text

Programming doesn't just concern numbers. Text is one of the most common types of values used in programs.

A snippet of text is represented by a **string value** in Python. The word "*string*" is a programming term for a sequence of characters. A string might contain a single character, a word, a

sentence, or a whole book.

To distinguish text data from actual code, we demarcate strings by putting quotation marks around them. Single quotes (`'`) and double quotes (`"`) are both valid, but the types of opening and closing quotation marks must match. The contents can be any sequence of characters, including numbers and symbols.

We've seen strings before in `print` statements. Below, two different strings are passed as arguments to the `print` function.

In [2]:
```
print("I <3", 'Data Science')
```

```
I <3 Data Science
```

Just like names can be given to numbers, names can be given to string values. The names and strings aren't required to be similar in any way. Any name can be assigned to any string.

In [3]:
```
one = 'two'
plus = '*'
print(one, plus, one)
```

```
two * two
```

Question 2. Yuri Gagarin was the first person to travel through outer space. When he emerged from his capsule upon landing on Earth, he reportedly had the following conversation with a woman and girl who saw the landing:

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

The cell below contains unfinished code. Fill in the `...` s so that it prints out this conversation *exactly* as it appears above.

In [4]:
```
woman_asking = 'The woman asked:'
woman_quote = '"Can it be that you have come from outer space?"'
gagarin_reply = 'Gagarin replied:'
gagarin_quote = '"As a matter of fact, I have!"'

print(woman_asking, woman_quote)
print(gagarin_reply, gagarin_quote)
```

```
The woman asked: "Can it be that you have come from outer space?"
Gagarin replied: "As a matter of fact, I have!"
```

2.1. String Methods

Strings can be transformed using **methods**, which are functions that involve an existing string and some other arguments. One example is the `replace` method, which replaces all instances of some part of a string with some alternative.

A method is invoked on a string by placing a `.` after the string value, then the name of the method, and finally parentheses containing the arguments. Here's a sketch, where the `<` and `>` symbols aren't part of the syntax; they just mark the boundaries of sub-expressions.

```
<expression that evaluates to a string>.<method name>(<argument>,
<argument>, ...)
```

Try to predict the output of these examples, then execute them.

In [5]:
```
# Replace one letter
'Hello'.replace('o', 'a')
```

Out[5]:
```
'Hella'
```

In [6]:
```
# Replace a sequence of letters, which appears twice
'hitchhiker'.replace('hi', 'ma')
```

Out[6]:
```
'matchmaker'
```

Once a name is bound to a string value, methods can be invoked on that name as well. The name is still bound to the original string, so a new name is needed to capture the result.

In [7]:
```
sharp = 'edged'
hot = sharp.replace('ed', 'ma')
print('sharp:', sharp)
print('hot:', hot)
```

```
sharp: edged
hot: magma
```

You can call functions on the results of other functions. For example,

```
max(abs(-5), abs(3))
```

has value 5. Similarly, you can invoke methods on the results of other method (or function) calls.

In [52]:
```
# Calling replace on the output of another call to
# replace
'train'.replace('t', 'ing').replace('in', 'de')
```

Out[52]:
```
'degrade'
```

Here's a picture of how Python evaluates a "chained" method call like that:

Question 3. Assign strings to the names `you` and `this` so that the final expression evaluates to a 10-letter English word with three double letters in a row. Click this link if you want a hint.

Hint: After you guess at some values for `you` and `this`, it's helpful to see the value of the variable `the`. Try printing the value of `the` by adding a line like this:

```
print(the)
```

In [53]:
```
you = 'keep'
this = 'book'
a = 'beeper'
the = a.replace('p', you)
the.replace('bee', this)
```

Out[53]:
```
'bookkeeper'
```

Other string methods do not take any arguments at all, because the original string is all that's needed to compute the result. In this case, parentheses are still needed, but there's nothing in between the parentheses. Here are some methods that work that way:

Method name	Value
`lower`	a lowercased version of the string
`upper`	an uppercased version of the string
`capitalize`	a version with the first letter capitalized
`title`	a version with the first letter of every word capitalized

In [10]:
```
'yALe universITY'.title()
```

Out[10]:
```
'Yale University'
```

All these string methods are useful, but most programmers don't memorize their names or how to use them. In the "real world," people usually just search the internet for documentation and examples. A complete list of string methods appears in the Python language documentation. Stack Overflow has a huge database of answered questions that often demonstrate how to use these methods to achieve various ends.

2.2. Converting to and from Strings

Strings and numbers are different *types* of values, even when a string contains the digits of a number. For example, evaluating the following cell causes an error because an integer cannot be added to a string.

In [11]:
```
#8 + "8" #commented out code; gives error
```

However, there are built-in functions to convert numbers to strings and strings to numbers.

```
int:   Converts a string of digits to an integer ("int") value
float: Converts a string of digits, perhaps with a decimal point,
to a decimal ("float") value
str:   Converts any value to a string
```

Try to predict what the following cell will evaluate to, then evaluate it.

In [12]:
```
8 + int("8")
```

Out[12]: 16

Suppose you're writing a program that looks for dates in a text, and you want your program to find the amount of time that elapsed between two years it has identified. It doesn't make sense to subtract two texts, but you can first convert the text containing the years into numbers.

Question 4. Finish the code below to compute the number of years that elapsed between `one_year` and `another_year`. Don't just write the numbers `1618` and `1648` (or `30`); use a conversion function to turn the given text data into numbers.

In [13]:
```
# Some text data:
one_year = "2012"
another_year = "2016"

# Complete the next line.  Note that we can't just write:
#   another_year - one_year
# If you don't see why, try seeing what happens when you
# write that here.
difference = int("2016")-int("2012")
difference
```

Out[13]: 4

2.3. Strings as function arguments

String values, like numbers, can be arguments to functions and can be returned by functions. The function `len` takes a single string as its argument and returns the number of characters in the string: its **len**-gth.

Note that it doesn't count *words*. `len("one small step for man")` is 22, not 5.

Question 5. Use `len` to find out the number of characters in the very long string in the next cell. (It's the first sentence of the English translation of the French Declaration of the Rights of Man.) The length of a string is the total number of characters in it, including things like spaces and punctuation. Assign `sentence_length` to that number.

In [14]:
```
a_very_long_sentence = "The representatives of the French people, organized as a
sentence_length = len(a_very_long_sentence)
sentence_length
```

Out[14]: 896

3. Importing code

> What has been will be again,
> what has been done will be done again;
> there is nothing new under the sun.

Most programming involves work that is very similar to work that has been done before. Since writing code is time-consuming, it's good to rely on others' published code when you can. Rather than copy-pasting, Python allows us to **import** other code, creating a **module** that contains all of the names created by that code.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example. The `math` module is extremely useful in computing mathematical expressions in Python.

Suppose we want to very accurately compute the area of a circle with radius 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module has `pi` defined for us:

In [15]:
```
import math
radius = 5
area_of_circle = radius**2 * math.pi
area_of_circle
```

Out[15]:
```
78.53981633974483
```

`pi` is defined inside `math`, and the way that we access names that are inside modules is by writing the module's name, then a dot, then the name of the thing we want:

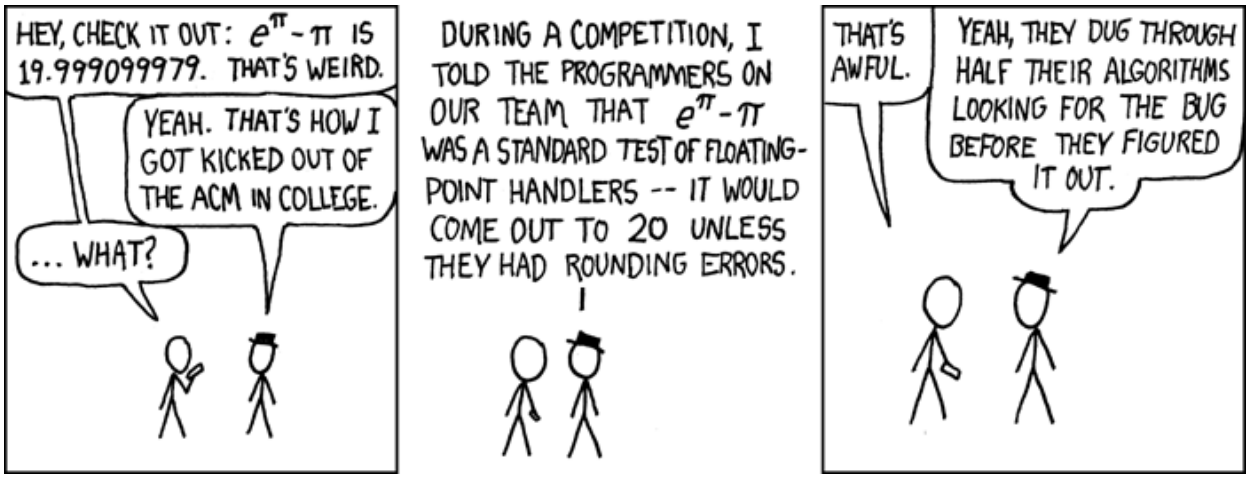
```
<module name>.<name>
```

In order to use a module at all, we must first write the statement `import <module name>`. That statement creates a module object with things like `pi` in it and then assigns the name `math` to that module. Above we have done that for `math`.

Question 6. `math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

In [16]:
```
near_twenty = ((math.e)**(math.pi))-math.pi
near_twenty
```

Out[16]:
```
19.99909997918947
```



3.1. Importing functions

Modules can provide other named things, including **functions**. For example, `math` provides the name `sin` for the sine function. Having imported `math` already, we can write `math.sin(3)` to compute the sine of 3. (Note that this sine function considers its argument to be in radians, not degrees. 180 degrees are equivalent to π radians.)

Question 7. A $\frac{\pi}{4}$-radian (45-degree) angle forms a right triangle with equal base and height, pictured below. If the hypotenuse (the radius of the circle in the picture) is 1, then the height is $\sin(\frac{\pi}{4})$. Compute that using `sin` and `pi` from the `math` module. Give the result the name `sine_of_pi_over_four`.

In [17]:
```
sine_of_pi_over_four = math.sin((math.pi)/4)
sine_of_pi_over_four
```

Out[17]:
```
0.7071067811865475
```

For your reference, here are some more examples of functions from the `math` module.

Note how different methods take in different number of arguments. Often, the documentation of the module will provide information on how many arguments is required for each method.

In [18]:
```
# Calculating factorials.
math.factorial(5)
```

Out[18]:
```
120
```

In [19]:
```
# Calculating logarithms (the logarithm of 8 in base 2).
# The result is 3 because 2 to the power of 3 is 8.
math.log(8, 2)
```

Out[19]:
```
3.0
```

In [20]:
```
# Calculating square roots.
math.sqrt(5)
```

Out[20]:
```
2.23606797749979
```

In [21]:
```
# Calculating cosines.
math.cos(math.pi)
```

Out[21]:
```
-1.0
```

There's many variations of how we can import methods from outside sources. For example, we can import just a specific method from an outside source, we can rename a library we import, and we can import every single method from a whole library.

In [22]:
```
#Importing just cos and pi from math.
#Notice that we don't have to use "math." before hand for cos and pi
from math import cos, pi
print(cos(pi))
#We do have to use it infront of other methods from math, though
math.log(pi)
```

```
-1.0
```

Out[22]:
```
1.1447298858494002
```

In [23]:
```
#We can nickname math as something else, if we don't want to type math
import math as m
m.log(m.pi)
```

Out[23]:
```
1.1447298858494002
```

In [24]:
```
#Lastly, we can import ever thing from math
from math import *
log(pi)
```

Out[24]:
```
1.1447298858494002
```

4. Arrays

Up to now, we haven't done much that you couldn't do yourself by hand, without going through the trouble of learning Python. Computers are most useful when you can use a small amount of code to *do the same action* to *many different things*.

For example, in the time it takes you to calculate the 18% tip on a restaurant bill, a laptop can calculate 18% tips for every restaurant bill paid by every human on Earth that day. (That's if you're pretty fast at doing arithmetic in your head!)

Arrays are how we put many values in one place so that we can operate on them as a group. For example, if `billions_of_numbers` is an array of numbers, the expression

```
.18 * billions_of_numbers
```

gives a new array of numbers that's the result of multiplying each number in `billions_of_numbers` by .18 (18%). Arrays are not limited to numbers; we can also put all the words in a book into an array of strings.

Concretely, an array is a **collection of values of the same type**, like a column in an Excel spreadsheet.

4.1. Making arrays

You can type in the data that goes in an array yourself, but that's not typically how programs work. Normally, we create arrays by loading them from an external source, like a data file.

Arrays are provided by a package called NumPy (pronounced "NUM-pie" or, if you prefer to pronounce things incorrectly, "NUM-pee"). The package is called `numpy`, but it's standard to rename it `np` for brevity. You can do that with:

```
import numpy as np
```

Execute the following cell so that all the names from the `numpy` module are available to you when you type `np`.

In [25]:
```
import numpy as np
```

Now, to create an array, call the function `array`. Each argument you pass to `array` will be in the array it returns. Run this cell to see an example:

In [26]:
```
sample_array = np.array([0.125, 4.75, -1.3])
sample_array
```

Out[26]:
```
array([ 0.125,  4.75 , -1.3  ])
```

Each value in an array (in the above case, the numbers 0.125, 4.75, and -1.3) is called an *element* of that array. Pay attention to how `array` requires you to first use parantheses to call the function, then brackets to call each element of the array.

Arrays themselves are also values, just like numbers and strings. That means you can assign them names or use them as arguments to functions.

Question 7 Make an array containing the numbers 1, 2, and 3, in that order. Name it `small_numbers`.

In [27]:
```
small_numbers = np.array([1,2,3])
```

```
small_numbers
```

Out[27]:
```
array([1, 2, 3])
```

Question 8. Make an array containing the numbers 0, 1, -1, π, and e, in that order. Name it `interesting_numbers`. *Hint:* How did you get the values π and e earlier? You can refer to them in exactly the same way here.

In [28]:
```
interesting_numbers = np.array([0,1,-1, math.pi, math.e])
interesting_numbers
```

Out[28]:
```
array([ 0.        ,  1.        , -1.        ,  3.14159265,  2.71828183])
```

Question 9. Make an array containing the five strings `"Hello"`, `","`, `" "`, `"world"`, and `"!"`. (The third one is a single space inside quotes.) Name it `hello_world_components`.

Note: If you print `hello_world_components`, you'll notice some extra information in addition to its contents: `dtype='<U5'`. That's just NumPy's extremely cryptic way of saying that the things in the array are strings.

In [29]:
```
hello_world_components = np.array(["Hello",","," ", "world","!"])
hello_world_components
```

Out[29]:
```
array(['Hello', ',', ' ', 'world', '!'], dtype='<U5')
```

4.1.1. `np.arange`

Very often in data science, we want to work with many numbers that are evenly spaced within some range. NumPy provides a special function for this called `arange`. `np.arange(start, stop, space)` produces an array with all the numbers starting at `start` and counting up by `space`, stopping before `stop` is reached.

For example, the value of `np.arange(1, 6, 2)` is an array with elements 1, 3, and 5 -- it starts at 1 and counts up by 2, then stops before 6. In other words, it's equivalent to `array([1, 3, 5])`.

`np.arange(4, 9, 1)` is an array with elements 4, 5, 6, 7, and 8. (It doesn't contain 9 because `np.arange` stops *before* the stop value is reached.)

Question 10. Use `np.arange` to create an array with the multiples of 99 from 0 up to (**and including**) 9999. (So its elements are 0, 99, 198, 297, etc.)

In [30]:
```
multiples_of_99 = np.arange(0,10000,99)
multiples_of_99
```

Out[30]:
```
array([   0,   99,  198,  297,  396,  495,  594,  693,  792,  891,  990,
       1089, 1188, 1287, 1386, 1485, 1584, 1683, 1782, 1881, 1980, 2079,
       2178, 2277, 2376, 2475, 2574, 2673, 2772, 2871, 2970, 3069, 3168,
       3267, 3366, 3465, 3564, 3663, 3762, 3861, 3960, 4059, 4158, 4257,
       4356, 4455, 4554, 4653, 4752, 4851, 4950, 5049, 5148, 5247, 5346,
```

```
5445, 5544, 5643, 5742, 5841, 5940, 6039, 6138, 6237, 6336, 6435,
6534, 6633, 6732, 6831, 6930, 7029, 7128, 7227, 7326, 7425, 7524,
7623, 7722, 7821, 7920, 8019, 8118, 8217, 8316, 8415, 8514, 8613,
8712, 8811, 8910, 9009, 9108, 9207, 9306, 9405, 9504, 9603, 9702,
9801, 9900, 9999])
```

4.2. Working with single elements of arrays ("indexing")

Let's work with a more interesting dataset. The next cell creates an array called `population` that includes estimated world populations in every year from **1950** to roughly the present. (The estimates come from the US Census Bureau website.)

Rather than type in the data manually, we've loaded them from a file on your computer called `world_population.csv` . You'll learn how to do that next week.

In [31]:
```
# Don't worry too much about what goes on in this cell.
from pandas import *
population = read_csv("world_population.csv", usecols = ["Population"])
population
```

Out[31]:

	Population
0	2557628654
1	2594939877
2	2636772306
3	2682053389
4	2730228104
...	...
61	6944055583
62	7022349283
63	7101027895
64	7178722893
65	7256490011

66 rows × 1 columns

Here's how we get the first element of `population` , which is the world population in the first year in the dataset, 1950.

In [32]:
```
population.iloc[0]
```

Out[32]:
```
Population    2557628654
Name: 0, dtype: int64
```

The value of that expression is the number 2557628654 (around 2.5 billion), because that's the first thing in the data frame `population` .

Notice that we wrote `.iloc[0]`, not `.iloc[1]`, to get the first element. This is a weird convention in computer science. 0 is called the *index* of the first item. It's the number of elements that appear *before* that item. So 3 is the index of the 4th item.

Here are some more examples. In the examples, we've given names to the things we get out of `population`. Read and run each cell.

In [33]:
```
# The third element in the array is the population
# in 1952.
population_1952 = population.iloc[2]
population_1952
```

Out[33]:
```
Population    2636772306
Name: 2, dtype: int64
```

In [34]:
```
# The thirteenth element in the array is the population
# in 1962 (which is 1950 + 12).
population_1962 = population.iloc[12]
population_1962
```

Out[34]:
```
Population    3140093217
Name: 12, dtype: int64
```

In [35]:
```
# The array has only 66 elements, so this doesn't work.
# (There's no element with 66 other elements before it.)
#population_2016 = population.iloc[66]
#population_2016
```

Question 11. Set `population_1973` to the world population in 1973, by getting the appropriate element from `population` using `iloc`.

In [36]:
```
population_1973 = population.iloc[23]
population_1973
```

Out[36]:
```
Population    3942096442
Name: 23, dtype: int64
```

4.3. Doing something to every element of an array

Arrays are primarily useful for doing the same operation many times, so we don't often have to use `.iloc` and work with single elements.

Logarithms

Here is one simple question we might ask about world population:

> How big was the population in *orders of magnitude* in each year?

The logarithm function is one way of measuring how big a number is. The logarithm (base 10) of a number increases by 1 every time we multiply the number by 10. It's like a measure of how many decimal digits the number has, or how big it is in orders of magnitude.

We could try to answer our question like this, using the `log10` function from the `math` module and the `item` method you just saw:

In [37]:
```
import math

population_1950_magnitude = math.log10(population.iloc[0])
population_1951_magnitude = math.log10(population.iloc[1])
population_1952_magnitude = math.log10(population.iloc[2])
population_1953_magnitude = math.log10(population.iloc[3])
...
```

Out[37]: `Ellipsis`

But this is tedious and doesn't really take advantage of the fact that we are using a computer.

Instead, NumPy provides its own version of `log10` that takes the logarithm of each element of an array. It takes a single array of numbers as its argument. It returns an array of the same length, where the first element of the result is the logarithm of the first element of the argument, and so on.

In the below code, we are going to `apply` the function `np.log10` to every element in the data frame `population`. We do this by first calling the data frame `population`.

In [38]:
```
population_magnitudes = population.apply(np.log10)
population_magnitudes
```

Out[38]:

	Population
0	9.407837
1	9.414127
2	9.421073
3	9.428467
4	9.436199
...	...
61	9.841613
62	9.846482
63	9.851321
64	9.856047
65	9.860727

66 rows × 1 columns

Arithmetic

You can also do arithmetic on a data frame. For example, you can divide all the population numbers by 1 billion to get numbers in billions:

In [39]:
```
population_in_billions = population / 1000000000
population_in_billions
```

Out[39]:

	Population
0	2.557629
1	2.594940
2	2.636772
3	2.682053
4	2.730228
...	...
61	6.944056
62	7.022349
63	7.101028
64	7.178723
65	7.256490

66 rows × 1 columns

You can do the same with addition, subtraction, multiplication, and exponentiation (**). For example, you can calculate a tip on several restaurant bills at once (in this case just 3):

In [40]:
```
restaurant_bills = np.array([20.12, 39.90, 31.01])
print("Restaurant bills:\t", restaurant_bills)
tips = .2 * restaurant_bills
print("Tips:\t\t\t", tips)
```

```
Restaurant bills:	 [20.12 39.9  31.01]
Tips:			 [4.024 7.98  6.202]
```

Question 12. Suppose the total charge at a restaurant is the original bill plus the tip. That means we can multiply the original bill by 1.2 to get the total charge. Compute the total charge for each bill in `restaurant_bills` .

In [41]:
```
total_charges = restaurant_bills*1.2
total_charges
```

Out[41]:
```
array([24.144, 47.88 , 37.212])
```

Question 13. more_restaurant_bills.csv contains 100,000 bills! Compute the total charge for each one. How is your code different?

In [42]:
```
more_restaurant_bills = read_csv("more_restaurant_bills.csv")
more_total_charges = more_restaurant_bills*1.2
more_total_charges
```

Out[42]:

	Bill
0	20.244
1	20.892
2	12.216
3	11.808
4	22.128
...	...
99995	35.736
99996	7.440
99997	19.308
99998	18.336
99999	35.664

100000 rows × 1 columns

Question 14. What is the sum of all the bills in `more_restaurant_bills`, *including tips*?

We have not yet covered this function. Learning how to get comfortable Googling for help is an important step in learning how to program. See if you can figure this out on your own.

Hint: Remember `NumPy`.

In [43]:
```
sum_of_bills = np.sum(more_total_charges)
sum_of_bills
```

Out[43]:
```
Bill    1795730.064
dtype: float64
```

5. Merging Voting Data

I created fake data reflecting how Obama/Romney and Trump/Clinton performed in 10 Connecticut precincts. See below.

In [44]:
```
ct12 = read_csv("election12.csv")
ct16 = read_csv("election16.csv")
```

In [45]:
```
# Here is the 2012 data
ct12
```

Out[45]:

	precinct	Romney	Obama
0	CT1	82	167
1	CT2	165	198
2	ct3	154	88

	precinct	Romney	Obama
3	CT4	77	77
4	CT5	198	121
5	CT-6	165	176
6	CT7	132	209
7	CT8	198	132
8	CT9	165	187
9	CT10	198	66

In [46]:
```
# Here is the 2016 data
ct16
```

Out[46]:

	Precinct	Trump	Clinton
0	CT1	88	154
1	CT2	99	165
2	CT3	165	165
3	CT4	132	121
4	CT5	165	110
5	CT6	110	88
6	CT7	154	77
7	CT8	77	121
8	CT9	165	154
9	CT10	77	99

Question 14.

What percent of the vote did Romney receive in each precinct? To do this, create a new column called romney_vote_share which is defined as Romney / (Romney + Obama). Do the same with Trump.

Below is some code to help you get started.

In [47]:
```
ct12['romney_vote_share'] = ct12['Romney'] /(ct12['Romney']+ct12['Obama'])# You
ct12 # Check your work
```

Out[47]:

	precinct	Romney	Obama	romney_vote_share
0	CT1	82	167	0.329317
1	CT2	165	198	0.454545
2	ct3	154	88	0.636364
3	CT4	77	77	0.500000

	precinct	Romney	Obama	romney_vote_share
4	CT5	198	121	0.620690
5	CT-6	165	176	0.483871
6	CT7	132	209	0.387097
7	CT8	198	132	0.600000
8	CT9	165	187	0.468750
9	CT10	198	66	0.750000

In [48]:
```
ct16['trump_vote_share'] = ct16['Trump'] / (ct16['Trump']+ct16['Clinton']) # You
ct16 # Check your work
```

Out[48]:

	Precinct	Trump	Clinton	trump_vote_share
0	CT1	88	154	0.363636
1	CT2	99	165	0.375000
2	CT3	165	165	0.500000
3	CT4	132	121	0.521739
4	CT5	165	110	0.600000
5	CT6	110	88	0.555556
6	CT7	154	77	0.666667
7	CT8	77	121	0.388889
8	CT9	165	154	0.517241
9	CT10	77	99	0.437500

Question 15 (challenging).

Join together the two election years so that you have one table that has precinct, trump_vote_share, and romney_vote_share. Use this to calculate a column called change_in_vote_share.

To do this, you will first need to clean the ct12 data. Notice a few things:

- In `CT12`, the column is called `precinct`. In `CT16`, it is called `Precinct`.
- In `CT12`, see `ct3` and `CT-6`. You will want these to match the same names as `CT16`.

In [49]:
```
# Make your changes to ct12 here.
ct12.columns=['Precinct', 'Romney', 'Obama', 'romney_vote_share']
ct12["Precinct"]=ct12["Precinct"].replace("ct3", "CT3")
ct12["Precinct"]=ct12["Precinct"].replace("CT-6","CT6")
# First, rename your column.
# For a hint, see https://www.interviewqs.com/ddi_code_snippets/rename_columns

# Second, fix ct3 and CT-6.
# You will need something like: data["name"]= data["name"].replace("Josh", "Josh
```

```
# You can read more at https://www.geeksforgeeks.org/python-pandas-series-str-re

# Check your work
ct12
```

Out[49]:

	Precinct	Romney	Obama	romney_vote_share
0	CT1	82	167	0.329317
1	CT2	165	198	0.454545
2	CT3	154	88	0.636364
3	CT4	77	77	0.500000
4	CT5	198	121	0.620690
5	CT6	165	176	0.483871
6	CT7	132	209	0.387097
7	CT8	198	132	0.600000
8	CT9	165	187	0.468750
9	CT10	198	66	0.750000

In [50]:

```
# Once CT12 Precinct matches CT16 Precinct, you can create a merged table.
# Use this code.
merged_elections = merge(ct12, ct16, on='Precinct')

# Look at the ouput.
merged_elections
```

Out[50]:

	Precinct	Romney	Obama	romney_vote_share	Trump	Clinton	trump_vote_share
0	CT1	82	167	0.329317	88	154	0.363636
1	CT2	165	198	0.454545	99	165	0.375000
2	CT3	154	88	0.636364	165	165	0.500000
3	CT4	77	77	0.500000	132	121	0.521739
4	CT5	198	121	0.620690	165	110	0.600000
5	CT6	165	176	0.483871	110	88	0.555556
6	CT7	132	209	0.387097	154	77	0.666667
7	CT8	198	132	0.600000	77	121	0.388889
8	CT9	165	187	0.468750	165	154	0.517241
9	CT10	198	66	0.750000	77	99	0.437500

In [51]:

```
# Create change_in_vote_share
merged_elections["change_in_vote_share"] = merged_elections["trump_vote_share"]

# Calculate the average (mean) of change in vote share
print(merged_elections.loc[:,"change_in_vote_share"].mean())
```

```
# Check your output
merged_elections
```

```
-0.030440577260069157
```

Out[51]:

	Precinct	Romney	Obama	romney_vote_share	Trump	Clinton	trump_vote_share	change_in_vo
0	CT1	82	167	0.329317	88	154	0.363636	
1	CT2	165	198	0.454545	99	165	0.375000	-
2	CT3	154	88	0.636364	165	165	0.500000	-
3	CT4	77	77	0.500000	132	121	0.521739	
4	CT5	198	121	0.620690	165	110	0.600000	-
5	CT6	165	176	0.483871	110	88	0.555556	
6	CT7	132	209	0.387097	154	77	0.666667	
7	CT8	198	132	0.600000	77	121	0.388889	
8	CT9	165	187	0.468750	165	154	0.517241	
9	CT10	198	66	0.750000	77	99	0.437500	-

Congratulations!

You are done with the lab. Before you finish and submit, please fill out this brief evaluation:

- I spent around 6 hours on this lab.
- This lab was just about the right difficulty.

To turn in your lab, you will need to submit a PDF through Canvas. You can download a notebook by opening it, turning Edit mode on, then navigating to File -> Download as -> PDF.