Learin_Python-Round2

February 22, 2019

1 Teaching Yourself Python Basics

1.1 Intro

The Coursera course taught by the University of Michigan wasn't really doing it for me. So I decided to start from scratch with this handy notebook, where I will lay down the Python basics to remind myself (and whoever else may be interested) how things work. Ideally this will help in the longrun when taking the upper level courses for the University of Michigan's Data Science with Python (especially because they don't go into too much detail in the course). My work here will be based off of the lessons on the website www.learnpython.org. So without further ado, let's get started!

Aside:

For new users checking this notebook out, if you would like to play with it in dark mode (rather than the bright default Jupyter offers us), run the below cell and reboot Jupyter.

```
In [ ]: !pip install jupyterthemes
        !jt -t chesterish
```

1.2 Learning the Basics

1.2.1 Hello World!

Let's start from the absolute bottom up so that absolutely no stone is left unturned. We'll do this by opening up our window, looking outside, and giving a hearty "Hello World!"

```
In [13]: print("Hello World!")
```

```
Hello World!
```

Easy peasy. That print statement will do exactly what it says; print out what you put inside. Now unlike R, we don't need braces or anything around things like if statements. Instead they just need to be indented:

```
In [14]: x = 1
         if x == 1:
             # indented four spaces
             print("x is 1.")
```

```
x is 1.
```

1.2.2 Variables and Types

Python is object oriented, so luckily this part is pretty straight forward. I'll try to speed through this part.

In [15]:
```
### Numbers:

myint = 5
myfloat = 5.0 #or
myfloat2_thefloatening = float(5)

print(type(myint))
print(type(myfloat))
print(type(myfloat2_thefloatening))
```

```
<class 'int'>
<class 'float'>
<class 'float'>
```

In [16]:
```
### Strings:

howdy = 'hello!'
nihao = "hello!"

# Notice both '' and "" will work when making strings.
# Just be aware to use "" if you have apostrophes.

print(howdy)
print(nihao)
```

```
hello!
hello!
```

In [17]:
```
### None:
depression = None
# Explains itself
type(depression)
```

Out[17]: NoneType

In [18]:
```
### Variable Operations:

one = 1
two = 2
three = one + two
print(three)
```

```
hello = "hello"
world = "world"
helloworld = hello + " " + world
print(helloworld)

# We can even assign multiple variables at once

a, b = 3, 4
print(a,b)
```

```
3
hello world
3 4
```

It's important to note that mixing operations between numbers and strings won't work.

```
In [19]: # This will not work!
         one = 1
         two = 2
         hello = "hello"

         print(one + two + hello)
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-19-fa035bb013ee> in <module>()
          4 hello = "hello"
          5
    ----> 6 print(one + two + hello)


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

That being said, we can convert numbers into strings to accomplish this task!

```
In [20]: # This will work!
         one = 1
         two = 2
         three = str(one + two)

         print(three)
         print('h' + three + 'llo')
```

```
3
h3llo
```

Exercise The target of this exercise is to create a string, an integer, and a floating point number. The string should be named `mystring` and should contain the word "hello". The floating point number should be named `myfloat` and should contain the number 10.0, and the integer should be named `myint` and should contain the number 20. Easy right? Solution is below:

```
In [21]: mystring = 'hello'
         myfloat = float(10)
         myint = 20

         # testing code
         if mystring == "hello":
             print("String: %s" % mystring)
         if isinstance(myfloat, float) and myfloat == 10.0:
             print("Float: %f" % myfloat)
         if isinstance(myint, int) and myint == 20:
             print("Integer: %d" % myint)
```

```
String: hello
Float: 10.000000
Integer: 20
```

1.2.3 Lists

Lists are sort of like vectors in R or arrays in other languages. They contain any type of variable, and can contain as many variables as your PC can handle. Here's how to build an easy starter list.

```
In [22]: mylist = []
         mylist.append(1)
         mylist.append(2)
         mylist.append(3)
         print(mylist[0]) # prints 1
         print(mylist[1]) # prints 2
         print(mylist[2]) # prints 3
```

```
1
2
3
```

```
In [23]: # prints out 1,2,3
         for x in mylist:
             print(x)
```

```
1
2
3
```

You can also make a list in one single lined statement such as the following.

```
In [24]: mylist = [1,2,3]
```

Accessing an index which does not exist generates an exception (an error).

```
In [25]: mylist = [1,2,3]
         print(mylist[10])
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-25-ac9eeac6db06> in <module>()
      1 mylist = [1,2,3]
----> 2 print(mylist[10])


IndexError: list index out of range
```

Exercise In this exercise, you will need to add numbers and strings to the correct lists using the "append" list method. You must add the number 3 to the "numbers" list, and the word 'world' to the strings variable.

You will also have to fill in the variable second_name with the second name in the names list, using the brackets operator `[]`. **Note that the index is zero-based, so if you want to access the second item in the list, its index will be 1**.

```
In [26]: numbers = [1,2]
         strings = ['hello']
         names = ["John", "Eric", "Jessica"]

         # write your code here
         second_name = names[1]
         numbers.append(3)
         strings.append('world')

         # this code should write out the filled arrays
         # and the second name in the names list (Eric).
         print(numbers)
         print(strings)
         print("The second name on the names list is %s" % second_name)
```

```
[1, 2, 3]
['hello', 'world']
The second name on the names list is Eric
```

1.2.4 Basic Operators

We've touched on a few simple operations so far, so let's dive a little further in now.

Arithmetic Operators These are the ones we should all be familiar with, the mathematical operators of addition, subtraction, multiplication, and division. Don't forget when using these to keep PEMDAS in mind! That is, keep in mind your order of operations, as Python will follow it.

```
In [15]: number = 1 + (2 * 3 / 4.0)
         print(number)
```

```
2.5
```

A more complicated operation is the modulo operator (%) which returns the integer remainder of the division of two numbers: dividend % divisor = remainder.

```
In [16]: remainder = 11 % 3
         print(remainder)
```

```
2
```

Unlike languages like R that you know, are beautiful, Python doesn't always play nice with the human eye. Just like how Jupyter Notebook is a lesser version of RMarkdown...I'm getting off topic. So unlike what you might expect by saying 3^2 is 3^2 or "three squared", Python handles this with two multiplication symbols instead.

```
In [17]: squared = 3 ** 2
         cubed = 2 ** 3
         print(squared)
         print(cubed)
```

```
9
8
```

Using Operators with Lists Lists can be handles with operators as well. For example, you can combine lists by using the addition operator.

```
In [18]: even_numbers = [2,4,6,8]
         odd_numbers = [1,3,5,7]
         all_numbers = odd_numbers + even_numbers
         print(all_numbers)
```

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

Keep in mind that also unlike vectors in R, when multiplying a list by a scalar value, Python does **not** do vector algebra. Hence we get the following.

```
In [19]: list1 = [1,2,3]
         list2 = list1 * 3

         print(list1)
         print(list2)
```

```
[1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Exercise The target of this exercise is to create two lists called `x_list` and `y_list`, which contain 10 instances of the variables `x` and `y`, respectively. You are also required to create a list called `big_list`, which contains the variables `x` and `y`, 10 times each, by concatenating the two lists you have created.

```
In [20]: x = object()
         y = object()

         # TODO: change this code
         x_list = [x]
         x_list = x_list * 10
         y_list = [y]
         y_list = y_list * 10
         big_list = x_list + y_list

         print("x_list contains %d objects" % len(x_list))
         print("y_list contains %d objects" % len(y_list))
         print("big_list contains %d objects" % len(big_list))

         # testing code
         if x_list.count(x) == 10 and y_list.count(y) == 10:
             print("Almost there...")
         if big_list.count(x) == 10 and big_list.count(y) == 10:
             print("Great!")
```

```
x_list contains 10 objects
y_list contains 10 objects
big_list contains 20 objects
Almost there...
Great!
```

1.2.5 String Formatting

If you're familiar with C, you're in luck! Python uses C-style string formatting to create new, formatted strings. It's also similar to the `sprintf()` function in R, that allows the user to use C-style string formatting commands. (Have you noticed I sprinkle a lot of R in here? It's my baby.) The % operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like %s and %d. Here's an example.

```
In [21]: # This prints out "Hello, John!"
         name = "John"
         print("Hello, %s!" % name)
```

```
Hello, John!
```

To use two or more argument specifiers, use a tuple (parentheses).

```
In [22]: # This prints out "John is 23 years old."
         name = "John"
         age = 23
         print("%s is %d years old." % (name, age))
```

```
John is 23 years old.
```

Any object which is not a string can be formatted using the %s operator as well. The string which returns from the "repr" method of that object is formatted as the string. For example:

```
In [23]: # This prints out: A list: [1, 2, 3]
         mylist = [1,2,3]
         print("A list: %s" % mylist)
```

```
A list: [1, 2, 3]
```

Here are some basic argument specifiers we should know:
%s - String (or any object with a string representation, like numbers)
%d - Integers
%f - Floating point numbers
%.<number of digits>f - Floating point numbers with a fixed amount of digits to the right of the dot.
%x/%X - Integers in hex representation (lowercase/uppercase)

Exercise You will need to write a format string which prints out the data using the following syntax: `Hello John Doe. Your current balance is $53.44.`

```
In [24]: data = ("John", "Doe", 53.44)
         format_string = "Hello %s %s. Your current balance is $%.2f"

         print(format_string % data)
```

```
Hello John Doe. Your current balance is $53.44
```

1.2.6 Basic String Operations

By now we aught to know what strings are, but there's quite a big more we can do with them. To start, check out the `len()` function.

```
In [25]: astring = 'Zoinks!'
         len(astring)
```

```
Out[25]: 7
```

As you can see here the `len()` function returns 7 since that's how long the `astring` object is, including the punctuation. If we had spaces, those would be counted as well. We can also get a bit more precise with our string operations. What we're about to dive into can be useful for text mining.

```
In [26]: astring = "Zoinko the Clown"
         print(astring.index("o"))
```

```
1
```

That prints out 1, because the location of the first occurrence of the letter "o" is 1 characters away from the first character. Notice how there are actually three o's in the phrase - this method only recognizes the first.

But why didn't it print out 2? Isn't "o" the second character in the string? As we've mentioned before, Python (but not R because it's way cooler) start things at 0 instead of 1. So the index of "o" is 1.

On the flip side of this, if we used `.count` instead of `.index` we get the following.

```
In [27]: astring = "Zoinko the Clown"
         print(astring.count("o"))
```

```
3
```

As we see here, `.count` returns to us the number of times that the input character was used in the string. Say we wanted to take a slice of a string now. By that I mean, consider a situation where we only want a specific portion of a string. The code cell below shows how we may do this. Note that in this code cell, since the text is fairly long, we can make it into a multi-line string by coding as we do below.

```
In [48]: astring = ("Help me I'm trapped inside this computer!"
                    " This is not a joke please send help!")
         print(astring)
         print(astring[42:50] + astring[54:60])
```

```
Help me I'm trapped inside this computer! This is not a joke please send help!
This is a joke
```

Note that this uses the standard indexing methods that we should be getting used to (starting with 0 instead of 1).

We can also slice text with negative numbered index values, ie. if you were to write `astring[-3]` the print statement would return the 3rd character from the end. Another option we have comes when we put a 3rd item into the brackets, ie. `astring[x:y:z]`. Here the form is [start:stop:step], basically meaning that you start with index x, stop at index y, and go up by a value of z. Here's an example.

```
In [29]: astring = ("Help me I'm trapped inside this computer!"
                    " This is not a joke please send help!")
         print(astring[3:60:2])

pm ' rpe nieti optr hsi o  oe
```

In order to reverse a string we can do the following:

```
In [30]: astring = "Coding is cool...if you're a nerd."
         print(astring[::-1])

.dren a er'uoy fi...looc si gnidoC
```

Take that bully. We can also do this which gives off two fairly different impressions:

```
In [31]: astring = "Shut Up Dad"
         print(astring.upper())
         print(astring.lower())

SHUT UP DAD
shut up dad
```

And we can test what is contained within string values:

```
In [32]: astring = "Hello friends!"
         print(astring.startswith("Hello"))
         print(astring.endswith("asdfasdfasdf"))

True
False
```

The last thing we'll go over in this section is how to split a string into multiple strings grouped together in a list. This could be useful when doing text mining on down the line.

```
In [33]: astring = "We're almost done!"
         afewwords = astring.split(" ")
         afewwords

Out[33]: ["We're", 'almost', 'done!']
```

Exercise Try to fix the code to print out the correct information by changing the string. The Solution is done below.

```
In [34]: s = "Hey there! what should this string be?"
         # Length should be 20
         print("Length of s = %d" % len(s))

         # First occurrence of "a" should be at index 8
         print("The first occurrence of the letter a = %d" % s.index("a"))

         # Number of a's should be 2
         print("a occurs %d times" % s.count("a"))

         # Slicing the string into bits
         print("The first five characters are '%s'" % s[:5]) # Start to 5
         print("The next five characters are '%s'" % s[5:10]) # 5 to 10
         print("The thirteenth character is '%s'" % s[12]) # Just number 12
         print("The characters with odd index are '%s'" %s[1::2]) #(0-based indexing)
         print("The last five characters are '%s'" % s[-5:]) # 5th-from-last to end

         # Convert everything to uppercase
         print("String in uppercase: %s" % s.upper())

         # Convert everything to lowercase
         print("String in lowercase: %s" % s.lower())

         # Check how a string starts
         if s.startswith("Str"):
             print("String starts with 'Str'. Good!")

         # Check how a string ends
         if s.endswith("ome!"):
             print("String ends with 'ome!'. Good!")

         # Split the string into three separate strings,
         # each containing only a word
         print("Split the words of the string: %s" % s.split(" "))
```

```
Length of s = 38
The first occurrence of the letter a = 13
a occurs 1 times
The first five characters are 'Hey t'
The next five characters are 'here!'
The thirteenth character is 'h'
The characters with odd index are 'e hr!wa hudti tigb?'
The last five characters are 'g be?'
String in uppercase: HEY THERE! WHAT SHOULD THIS STRING BE?
String in lowercase: hey there! what should this string be?
Split the words of the string: ['Hey', 'there!', 'what', 'should', 'this', 'string', 'be?']
```

```
In [35]: s = "Strings are awesome!"
```

```python
        # Length should be 20
        print("Length of s = %d" % len(s))

        # First occurrence of "a" should be at index 8
        print("The first occurrence of the letter a = %d" % s.index("a"))

        # Number of a's should be 2
        print("a occurs %d times" % s.count("a"))

        # Slicing the string into bits
        print("The first five characters are '%s'" % s[:5]) # Start to 5
        print("The next five characters are '%s'" % s[5:10]) # 5 to 10
        print("The thirteenth character is '%s'" % s[12]) # Just number 12
        print("The characters with odd index are '%s'" %s[1::2]) #(0-based indexing)
        print("The last five characters are '%s'" % s[-5:]) # 5th-from-last to end

        # Convert everything to uppercase
        print("String in uppercase: %s" % s.upper())

        # Convert everything to lowercase
        print("String in lowercase: %s" % s.lower())

        # Check how a string starts
        if s.startswith("Str"):
            print("String starts with 'Str'. Good!")

        # Check how a string ends
        if s.endswith("ome!"):
            print("String ends with 'ome!'. Good!")

        # Split the string into three separate strings,
        # each containing only a word
        print("Split the words of the string: %s" % s.split(" "))
```

```
Length of s = 20
The first occurrence of the letter a = 8
a occurs 2 times
The first five characters are 'Strin'
The next five characters are 'gs ar'
The thirteenth character is 'a'
The characters with odd index are 'tig r wsm!'
The last five characters are 'some!'
String in uppercase: STRINGS ARE AWESOME!
String in lowercase: strings are awesome!
String starts with 'Str'. Good!
String ends with 'ome!'. Good!
Split the words of the string: ['Strings', 'are', 'awesome!']
```

1.2.7 Conditions

Like most programming languages, Python uses boolean variables to evaluate conditions (ie. True, False). Python will return these variables when a conditional statement is evaluated.

```
In [36]: x = 2
         print(x == 2) # prints out True
         print(x == 3) # prints out False
         print(x < 3) # prints out True
```

```
True
False
True
```

Boolean Operators Boolean operators allow for more complex Boolean expressions. The first examples of this we'll look at are "and" and "or".

```
In [39]: name = "Jacob"
         age = 24

         if name == "Jacob" and age == 24:
             print("Your name is Jacob, and you are also 24 years old.")

         if name == "Jacob" or name == "Ryan":
             print("Your name is either Jacob or Ryan.")

         if name == "Kevin" or name == "Jasper":
             print("Your name is Kasper.")
```

```
Your name is Jacob, and you are also 24 years old.
Your name is either Jacob or Ryan.
```

The next operator we'll discuss here is the "in" operator. This can be used to check if specific objects exist inside of an iterable object container, like a list.

```
In [40]: name = "Jacob"
         name_list = ["Jacob","Ryan"]

         if name in name_list:
             print("Your name is either Jacob or Ryan.")

         if name in ["Bill Gates","Elon Musk"]:
             print("Steal their wallet.")
```

```
Your name is either Jacob or Ryan.
```

One nice thing about Python is that instead of using brackets or something like that to define code blocks, it uses indentation. While this might seem strange, it actually makes for nicer looking code (which in turn is a little easier to read). The standard Python indentation is 4 spaces, although tabs and any other space size will work, as long as it is consistent. Notice that code blocks do not need any termination. So instead of having relatively ugly looking code like this:

```
In [41]: if <statement is="" true="">:
             <do something="">
             ....
             ....
         elif <another statement="" is="" true="">: # else if
             <do something="" else="">
             ....
             ....
         else:
             <do another="" thing="">
             ....
             ....
         </do></do></another></do></statement>
```

```
  File "<ipython-input-41-091a22c48e42>", line 1
    if <statement is="" true="">:
       ^
SyntaxError: invalid syntax
```

We get nicer code that looks like this:

```
In [43]: x = 2
         if x == 2:
             print("x equals two!")
         else:
             print("x does not equal to two.")

         y = 5
         if y == 2:
             print("y equals two!")
         else:
             print("y does not equal to two.")
```

```
x equals two!
y does not equal to two.
```

A statement is evaulated as true if one of the following is correct: - The "True" boolean variable is given, or calculated using an expression, such as an arithmetic comparison. - An object which is not considered "empty" is passed.

Here are some examples for objects which are considered as empty: - An empty string: "" - An empty list: [] - The number zero: 0 - The false boolean variable: False

Next up we'll talk a little bit about the "is" operator. While the == operator calculates whether or not a variable is equal to another, matching the values of the variables, the "is" operator matches the instances themselves. Most of the time we will find ourselves using the == operator instead, but it's important to know that we have this option as well. Below are some examples:

```
In [47]: x = [1,2,3]
         tx = type(x)
         y = [1,2,3]
         ty = type(y)
         print(x == y) # Prints out True
         print(x is y) # Prints out False
         print(x is x) # Prints out True
         print(tx is ty) # Prints out True
```

```
True
False
True
True
```

The last Boolean operator we'll discuss in this section is the "not" operator. Whereas most programming lanuages use `!` for indicating the inverse of a Boolean statement, Python uses the actual word `not`.

```
In [48]: print(not False) # Prints out True
         print((not False) == (False)) # Prints out False
```

```
True
False
```

Exercise Change the variables in the first section, so that each if statement resolves as True. The solution is in the second cell below.

```
In [49]: # change this code
         number = 10
         second_number = 10
         first_array = []
         second_array = [1,2,3]

         if number > 15:
             print("1")

         if first_array:
             print("2")

         if len(second_array) == 2:
```

```
        print("3")

    if len(first_array) + len(second_array) == 5:
        print("4")

    if first_array and first_array[0] == 1:
        print("5")
```

```
In [50]: # change this code
         number = 20
         second_number = 10
         first_array = [1,2,3]
         second_array = [1,2]

         if number > 15:
             print("1")

         if first_array: # an empty list causes this statement to be passed
             print("2")

         if len(second_array) == 2:
             print("3")

         if len(first_array) + len(second_array) == 5:
             print("4")

         if first_array and first_array[0] == 1: # ie. if first_array isn't empty and
             print("5")                          # the first entry is 1
```

```
1
2
3
4
5
```

1.2.8 Loops

There are two types of loops in Python, both of which we'll go over here. Python uses "for" and "while" loops.

The "for" Loop For loops iterate over a given sequence. For example,

```
In [54]: primes = [2, 3, 5, 7]
         for prime in primes:
             print(prime)

         names = ['Ryan','Jacob','Eric']
```

```
for i in names:
    print(i)
```

```
2
3
5
7
Ryan
Jacob
Eric
```

We can also use the `range()` function to iterate over a sequence of numbers. Note, the `range()` function returns a ne list with numbers of the specified range. Also keep in mind that this function is 0 based (meaning that it's indexed 0,1,2...).

```
In [60]: # Prints out the numbers 0,1,2,3,4
         for x in range(5):
             print(x)

         # Prints out 3,4,5
         for x in range(3, 6):
             print(x)

         # Prints out 3,5,7
         for x in range(3, 8, 2):
             print(x)
```

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

The "while" Loop While loops will repeat for as long as a certain Boolean condition is met. Be careful not to get yourself into infinite loops here! Here's an example:

```
In [62]: # Prints out 0,1,2,3,4

         count = 0            # Here we're initializing the variable
         while count < 5:     # that we'll iterate along inside the loop.
             print(count)
             count += 1  # This is the same as count = count + 1
```

```
0
1
2
3
4
```

"break" and "continue" Statements "break" is used to exit for or while loop. On the other hand, "continue" is used to skip the current block, and return to the "for" or "while" statement. As usual, here are some examples.

```
In [64]: # Prints out 0,1,2,3,4

         count = 0
         while True:
             print(count)
             count += 1
             if count >= 5:
                 break            # here we are giving the command
                                  # to break the loop and move on
         # Prints out only odd numbers - 1,3,5,7,9
         for x in range(10):
             # Check if x is even
             if x % 2 == 0:
                 continue        # here we're saying if x is even, skip past it.
             print(x)            # Otherwise, print x
```

```
0
1
2
3
4
1
3
5
7
9
```

What About "else"? In Python, we can use "else" for loops, unlike languages like C. When the loop condition of "for" or "while" statement fails then code part in "else" is executed. And similarly to our above explanations, if a "break" statement is executed inside the loop then the "else" is skipped; and even if there is a "continue" statement, the "else" part will be executed.

```
In [1]: # Prints out 0,1,2,3,4 and then it prints "count value reached 5"

        count=0
        while(count<5):
```

```
        print(count)
        count +=1
    else:
        print("count value reached %d" %(count))

    # Prints out 1,2,3,4
    for i in range(1, 10):
        if(i%5==0):
            break
        print(i)
    else:
        print(("this is not printed because for loop is terminated because of break"
               " but not due to fail in condition"))
```

```
0
1
2
3
4
count value reached 5
1
2
3
4
```

Exercise Loop through and print out all even numbers from the numbers list in the same order they are received. Don't print any numbers that come after 237 in the sequence. The solution is in the second code cell.

```
In [2]: numbers = [
            951, 402, 984, 651, 360, 69, 408, 319, 601, 485, 980, 507, 725, 547, 544,
            615, 83, 165, 141, 501, 263, 617, 865, 575, 219, 390, 984, 592, 236, 105, 942, 941
            386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953, 345,
            399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687, 217,
            815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 742, 717,
            958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380, 126, 721, 328, 753, 470
            743, 527
        ]

        # your code goes here
```

```
In [7]: numbers = [
            951, 402, 984, 651, 360, 69, 408, 319, 601, 485, 980, 507, 725, 547, 544,
            615, 83, 165, 141, 501, 263, 617, 865, 575, 219, 390, 984, 592, 236, 105, 942, 941
            386, 462, 47, 418, 907, 344, 236, 375, 823, 566, 597, 978, 328, 615, 953, 345,
            399, 162, 758, 219, 918, 237, 412, 566, 826, 248, 866, 950, 626, 949, 687, 217,
            815, 67, 104, 58, 512, 24, 892, 894, 767, 553, 81, 379, 843, 831, 445, 742, 717,
```

```
    958, 609, 842, 451, 688, 753, 854, 685, 93, 857, 440, 380, 126, 721, 328, 753, 470
    743, 527
]

for i in numbers:

    if i == 237:
        break

    if i % 2 == 1:
        continue

    print(i)
```

```
402
984
360
408
980
544
390
984
592
236
942
386
462
418
344
236
566
978
328
162
758
918
```

1.2.9 Functions

Functions are a nice way to divide and organize code into blocks. Functions often times make code more readable, and save time if we need to do a task multiple times. So how do we write functions in Python?

As we saw with loops, Python makes use of blocks and indentation to divide code. Below is an example of what a code block looks like (Ignore the #'s, they are being used to comment code out).

```
In [10]: #block_head:
         #    1st block line
```

```
#    2nd block line
#    ...
```

Functions in python are defined using the block keyword "def", followed with the function's name as the block's name. For example:

```
In [11]: def my_first_function():
             print("Hey mom look at me!")
```

Note that the function doesn't run until you call it:

```
In [12]: my_first_function()

Hey mom look at me!
```

Functions can also receive arguments, or variables that are passed through the caller to the function. Here is an example:

```
In [13]: def my_first_function_wargs(username, greeting):
             print("Hello, %s , From My Function!, I wish you %s" % (username, greeting))
```

Then we call a function with arguments similarly to above, but put our inputs in the parentheses in order.

```
In [14]: my_first_function_wargs("obewanjacobi","were dead")

Hello, obewanjacobi , From My Function!, I wish you were dead
```

Functions may also return a value to the caller. We do this by using the return statement.

```
In [15]: def sum_two_numbers(a, b):
             return a + b
```

Then similarly, we can run the function immediately and have it print the output, or we can save the output to a variable.

```
In [17]: # Run the statement only
         sum_two_numbers(2,3)

         # Return the output to the variable x
         x = sum_two_numbers(5,5)
```

Exercise In this exercise you'll use an existing function, and while adding your own to create a fully functional program.

- Add a function named `list_benefits()` that returns the following list of strings: "More organized code", "More readable code", "Easier code reuse", "Allowing programmers to share and connect code together"
- Add a function named `build_sentence(info)` which receives a single argument containing a string and returns a sentence starting with the given string and ending with the string " is a benefit of functions!"
- Run and see all the functions work together!

The solution is in the second code cell below.

In [19]:
```
# Modify this function to return a list of strings as defined above
def list_benefits():
    pass

# Modify this function to concatenate to each benefit
# - " is a benefit of functions!"
def build_sentence(benefit):
    pass

def name_the_benefits_of_functions():
    list_of_benefits = list_benefits()
    for benefit in list_of_benefits:
        print(build_sentence(benefit))
```

In [20]:
```
# Modify this function to return a list of strings as defined above
def list_benefits():
    benefits = ["More organized code", "More readable code",
                "Easier code reuse",
                "Allowing programmers to share and connect code together"]
    return benefits

# Modify this function to concatenate to each benefit
# - " is a benefit of functions!"
def build_sentence(benefit):
    return "%s is a benefit of functions!" % benefit

def name_the_benefits_of_functions():
    list_of_benefits = list_benefits()
    for benefit in list_of_benefits:
        print(build_sentence(benefit))

name_the_benefits_of_functions()
```

```
More organized code is a benefit of functions!
More readable code is a benefit of functions!
```

```
Easier code reuse is a benefit of functions!
Allowing programmers to share and connect code together is a benefit of functions!
```

1.2.10 Classes and Objects

Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.

A very basic class would look something like this:

```
In [1]: class MyClass:
            variable = "blah"

            def function(self):
                print("This is a message inside the class.")
```

We'll explain why you have to include that "self" as a parameter a little bit later. First, to assign the above class(template) to an object you would do the following:

```
In [2]: class MyClass:
            variable = "blah"

            def function(self):
                print("This is a message inside the class.")

        myobjectx = MyClass()
```

Now the variable "myobjectx" holds an object of the class "MyClass" that contains the variable and the function defined within the class called "MyClass". If this isn't making sense quite yet, fret not. More examples should clear things up in no time.

Accessing Object Variables Say you need a variable out of the class we just made above. To access the variable inside of the newly created object "myobjectx" you would do the following:

```
In [3]: class MyClass:
            variable = "blah"

            def function(self):
                print("This is a message inside the class.")

        myobjectx = MyClass()

        myobjectx.variable

Out[3]: 'blah'
```

As you can see, when we call the class, put in a period, and then call the variable we desire, it outputs that variable. Variables in classes can come in handy when you have a lot of variables and need to organize them in a clean fashion.

You can create multiple different objects that are of the same class(have the same variables and functions defined). However, each object contains independent copies of the variables defined in the class. For instance, if we were to define another object with the "MyClass" class and then change the string in the variable above:

```
In [4]: class MyClass:
            variable = "blah"

            def function(self):
                print("This is a message inside the class.")

        myobjectx = MyClass()
        myobjecty = MyClass()

        myobjecty.variable = "yackity"

        # Then print out both values
        print(myobjectx.variable)
        print(myobjecty.variable)
```

```
blah
yackity
```

In the above example, we changed the variable in the class by assigning it something new. But notice it didn't affect the class saved under `myobjectx`. It also didn't change the base `MyClass` class made, so if we were to make a `myobjectz = MyClass()`, we would still get the original class.

Accessing Object Functions To access a function inside of an object you use notation similar to accessing a variable:

```
In [5]: class MyClass:
            variable = "blah"

            def function(self):
                print("This is a message inside the class.")

        myobjectx = MyClass()

        myobjectx.function()
```

```
This is a message inside the class.
```

As you can see, this handles the same way as accessing a variable from a class does.

Exercise We have a class defined for vehicles. Create two new vehicles called car1 and car2. Set car1 to be a red convertible worth $60,000.00 with a name of Fer, and car2 to be a blue van named Jump worth $10,000.00. The solution will be in the second cell below.

In [6]:
```python
# define the Vehicle class
class Vehicle:
    name = ""
    kind = "car"
    color = ""
    value = 100.00
    def description(self):
        desc_str = ("%s is a %s %s worth $%.2f." %
                    (self.name, self.color, self.kind, self.value))
        return desc_str
# your code goes here

# test code
print(car1.description())
print(car2.description())
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-6-cd3d01fe16f8> in <module>()
     11
     12 # test code
---> 13 print(car1.description())
     14 print(car2.description())


NameError: name 'car1' is not defined
```

In [50]:
```python
# define the Vehicle class
class Vehicle:
    name = ""
    kind = "car"
    color = ""
    value = 100.00
    def description(self):
        desc_str = ("%s is a %s %s worth $%.2f." %
                    (self.name, self.color, self.kind, self.value))
        return desc_str
# your code goes here

car1 = Vehicle()
car1.name = 'Fer'
car1.kind = 'convertible'
car1.color = 'red'
car1.value = 60000.00
```

```
car2 = Vehicle()
car2.name = 'Jump'
car2.kind = 'van'
car2.color = 'blue'
car2.value = 10000.00

# test code
print(car1.description())
print(car2.description())
```

```
Fer is a red convertible worth $60000.00.
Jump is a blue van worth $10000.00.
```

1.2.11 Dictionaries

A dictionary is a data type in Python that is similar to an array, but instead of working with indexes, it works with keys and values. Each value stored in a dictionary can be accessed using a key, which is any type of object (a string, a number, a list, etc.) instead of using its index to address it.

For example, a database of phone numbers could be stored using a dictionary like this:

```
In [9]: phonebook = {} # initialize the dictionary
        phonebook["Frank"] = 5028477566
        phonebook["Stacy"] = 5028377264
        phonebook["Stacy's Mom"] = 5028675309
        print(phonebook)
```

```
{'Frank': 5028477566, 'Stacy': 5028377264, "Stacy's Mom": 5028675309}
```

Alternatively, a dictionary can be initialized with the same values in the following notation (this way you can make the whole dictionary in one step):

```
In [10]: phonebook = {
             "Frank" : 5028477566,
             "Stacy" : 5028377264,
             "Stacy's Mom" : 5028675309
         }
         print(phonebook)
```

```
{'Frank': 5028477566, 'Stacy': 5028377264, "Stacy's Mom": 5028675309}
```

Like lists, we can also iterate over a dictionary. However, a dictionary, unlike a list, does not keep the order of the values stored in it. To iterate over key value pairs, use the following syntax:

```
In [11]: phonebook = {
             "Frank" : 5028477566,
             "Stacy" : 5028377264,
             "Stacy's Mom" : 5028675309
         }
         for name, number in phonebook.items():
             print("Phone number of %s is %d" % (name, number))
```

```
Phone number of Frank is 5028477566
Phone number of Stacy is 5028377264
Phone number of Stacy's Mom is 5028675309
```

You can think of this as saying for each name and it's given number inside the phonebook (which are called its items), do the command in the loop. The variables `name` and `number` named in the for loop are just placeholder values to help us as users keep track of what's going on.

Say we want to remove a value from a dictionary. There are 2 ways to do this, both are demonstrated below:

```
In [16]: phonebook = {
             "Frank" : 5028477566,
             "Stacy" : 5028377264,
             "Stacy's Mom" : 5028675309
         }
         del phonebook["Frank"] # Because who needs a dude's phone number?
         print(phonebook)
```

```
{'Stacy': 5028377264, "Stacy's Mom": 5028675309}
```

```
In [17]: phonebook.pop("Stacy") # Stacy can't you see, you're just not the girl for me?
         print(phonebook)
```

```
{"Stacy's Mom": 5028675309}
```

And now we have all the numbers in our dictionary that really matter.

Exercise Add "Jake" to the phonebook with the phone number 5024152985, and remove Frank from the phonebook. The solution is in the second code cell below.

```
In [18]: phonebook = {
             "Frank" : 5028477566,
             "Stacy" : 5028377264,
             "Stacy's Mom" : 5028675309
         }

         # write your code here
```

```
# testing code
if "Jake" in phonebook:
    print("Jake is listed in the phonebook.")
if "Frank" not in phonebook:
    print("Frank is not listed in the phonebook.")
```

```
In [19]: phonebook = {
             "Frank" : 5028477566,
             "Stacy" : 5028377264,
             "Stacy's Mom" : 5028675309
         }

         # write your code here

         phonebook.pop("Frank")
         phonebook["Jake"] = 5024152985

         # testing code
         if "Jake" in phonebook:
             print("Jake is listed in the phonebook.")
         if "Frank" not in phonebook:
             print("Frank is not listed in the phonebook.")
```

```
Jake is listed in the phonebook.
Frank is not listed in the phonebook.
```

1.2.12 Modules and Packages

We did it, we made it to the last lesson under "**Learn the Basics**", feels like forever, doesn't it? This will be one of the more complex sections we go over. But no need to worry, that's why I wrote this little guy up. Hope this helps!

A module is a piece of software that has a specific functionality. For example, imagine you're building an app in Python. When doing this for example, you would have one module be responsible for the server, or what is calculated and run in the background. Then you would have another module to control the UI (user interface), and would control what is presented on screen. In this example, each module is a different file, and can be edited separately.

Writing Modules In this section we will give outlines and templates of how to write your own modules, but keep in mind that these modules don't work without some actual code in them. For that reason, to prevent from printing errors, we will simply leave the code commented out. If the code shows `##`, then it is an actual comment, whereas if the code shows `#`, then that's example code.

Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented. In the example of building an application, we will have two files, we will have:

```
In [20]: #myapp/            - the directory where the modules are stored
         #myapp/server.py   - the server module, what runs in the background
         #myapp/ui.py       - the UI module, controlling what is printed on screen
```

The Python script `server.py` will implement the app. It will use a function, maybe called `draw_app` from the file `ui.py`, or in other words, the `ui` module, that implements the logic for printing the app on the screen.

Modules are imported from other modules using the `import` command. In this example, the `server.py` script may look something like this:

```
In [21]: ## server.py
         ## import the ui module
         #import ui

         #def play_app():
         #    ...

         #def main():
         #    result = play_app()
         #    ui.ui_app(result)

         ## this means that if this script is executed, then
         ## main() will be executed
         #if __name__ == '__main__':
         #    main()
```

And the `ui` module may look something like this:

```
In [23]: ## ui.py

         #def ui_app():
         #    ...

         #def clear_screen(screen):
         #    ...
```

In this example, the `server` module imports the `load` module, which enables it to use functions implemented in that module. The `main` function would use the local function `play_app` to run the app, and then print the result of the app using a function implemented in the `ui` module called `ui_app`. To use the function `ui_app` from the `ui` module, we would need to specify in which module the function is implemented, using the dot operator. To reference the `ui_app` function from the `server` module, we would need to import the `ui` module and only then call `ui.ui_app()`.

When the `import ui` directive will run, the Python interpreter will look for a file in the directory which the script was executed from, by the name of the module with a `.py` suffix, so in our case it will try to look for `ui.py`. If it will find one, it will import it. If not, he will continue to look for built-in modules.

You may have noticed that when importing a module, a `.pyc` file appears, which is a compiled Python file. Python compiles files into Python bytecode so that it won't have to parse the files each time modules are loaded. If a `.pyc` file exists, it gets loaded instead of the `.py` file, but this process is transparent to the user.

Importing Module Objects to the Current Namespace We may also import the function `ui_app` directly into the main script's namespace, by using the `from` command.

```
In [1]: ## app.py
        ## import the ui module
        #from ui import ui_app

        #def main():
        #    result = play_app()
        #    ui_app(result)
```

You may have noticed that in this example, `ui_app` does not precede with the name of the module it is imported from, because we've specified the module name in the `import` command.

The advantages of using this notation is that it is easier to use the functions inside the current module because you don't need to specify which module the function comes from. However, any namespace cannot have two objects with the exact same name, so the `import` command may replace an existing object in the namespace.

Import all Objects From a Module We may also use the `import *` command to import all objects from a specific module, like this:

```
In [2]: ## app.py
        ## import the ui module
        #from ui import *

        #def main():
        #    result = play_app()
        #    ui_app(result)
```

This might be a bit risky as changes in the module might affect the module which imports it, but it is shorter and also does not require you to specify which objects you wish to import from the module.

Custom Import Name We may also load modules under any name we want. This is useful when we want to import a module conditionally to use the same name in the rest of the code.

For example, if you have two `ui` modules with slighty different names - you may do the following:

```
In [3]: ## app.py
        ## import the ui module
        #if visual_mode:
        #    # in visual mode, we print using graphics
        #    import ui_visual as ui
        #else:
        #    # in textual mode, we print out text
        #    import ui_textual as ui

        #def main():
```

```
#    result = play_app()
#    # this can either be visual or textual depending on visual_mode
#    ui.ui_app(result)
```

Module Initialization The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

This is useful to know, because this means that you can rely on this behavior for initializing objects. For example:

```
In [4]: ## ui.py

        #def ui_app():
        #    # when clearing the screen we can use the main
        #    # screen object initialized in this module
        #    clear_screen(main_screen)
        #    ...

        #def clear_screen(screen):
        #    ...

        #class Screen():
        #    ...

        ## initialize main_screen as a singleton
        #main_screen = Screen()
```

Extending Module Load Path There are a couple of ways we could tell the Python interpreter where to look for modules, aside from the default, which is the local directory and the built-in modules. You could either use the environment variable PYTHONPATH to specify additional directories to look for modules in, like this:

```
In [5]: #PYTHONPATH=/foo python app.py
```

```
  File "<ipython-input-5-a2d2c7aeb4ec>", line 1
    PYTHONPATH=/foo python app.py
               ^
SyntaxError: invalid syntax
```

This will execute app.py, and will enable the script to load modules from the foo directory as well as the local directory.

Another method is the sys.path.append function. You may execute it before running an import command:

```
In [6]: #sys.path.append("/foo")
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-6-cff76065f6a5> in <module>()
----> 1 sys.path.append("/foo")


NameError: name 'sys' is not defined
```

This will add the foo directory to the list of paths to look for modules in as well.

Exploring Built-In Modules Check out the full list of built-in modules in the Python standard library here: https://docs.python.org/3/library/.

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

If we want to import the module urllib, which enables us to create read data from URLs, we simply import the module:

```
In [8]: # import the library
        import urllib

        ## use it
        #urllib.urlopen(...)
```

We can look for which functions are implemented in each module by using the dir function:

```
In [9]: import urllib
        dir(urllib)

Out[9]: ['__builtins__',
         '__cached__',
         '__doc__',
         '__file__',
         '__loader__',
         '__name__',
         '__package__',
         '__path__',
         '__spec__',
         'error',
         'parse',
         'request',
         'response']
```

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter.

In [11]: help(urllib.parse)

```
Help on module urllib.parse in urllib:

NAME
    urllib.parse - Parse (absolute and relative) URLs.

DESCRIPTION
    urlparse module is based upon the following RFC specifications.

    RFC 3986 (STD66): "Uniform Resource Identifiers" by T. Berners-Lee, R. Fielding
    and L.  Masinter, January 2005.

    RFC 2732 : "Format for Literal IPv6 Addresses in URL's by R.Hinden, B.Carpenter
    and L.Masinter, December 1999.

    RFC 2396:  "Uniform Resource Identifiers (URI)": Generic Syntax by T.
    Berners-Lee, R. Fielding, and L. Masinter, August 1998.

    RFC 2368: "The mailto URL scheme", by P.Hoffman , L Masinter, J. Zawinski, July 1998.

    RFC 1808: "Relative Uniform Resource Locators", by R. Fielding, UC Irvine, June
    1995.

    RFC 1738: "Uniform Resource Locators (URL)" by T. Berners-Lee, L. Masinter, M.
    McCahill, December 1994

    RFC 3986 is considered the current standard and any future changes to
    urlparse module should conform with it.  The urlparse module is
    currently not entirely compliant with this RFC due to defacto
    scenarios for parsing, and for backward compatibility purposes, some
    parsing quirks from older RFCs are retained. The testcases in
    test_urlparse.py provides a good indicator of parsing behavior.

CLASSES
    DefragResult(builtins.tuple)
        DefragResult(DefragResult, _ResultMixinStr)
        DefragResultBytes(DefragResult, _ResultMixinBytes)
    ParseResult(builtins.tuple)
        ParseResult(ParseResult, _NetlocResultMixinStr)
        ParseResultBytes(ParseResult, _NetlocResultMixinBytes)
    SplitResult(builtins.tuple)
        SplitResult(SplitResult, _NetlocResultMixinStr)
        SplitResultBytes(SplitResult, _NetlocResultMixinBytes)
    _NetlocResultMixinBytes(_NetlocResultMixinBase, _ResultMixinBytes)
        ParseResultBytes(ParseResult, _NetlocResultMixinBytes)
        SplitResultBytes(SplitResult, _NetlocResultMixinBytes)
    _NetlocResultMixinStr(_NetlocResultMixinBase, _ResultMixinStr)
```

```
    ParseResult(ParseResult, _NetlocResultMixinStr)
    SplitResult(SplitResult, _NetlocResultMixinStr)
_ResultMixinBytes(builtins.object)
    DefragResultBytes(DefragResult, _ResultMixinBytes)
_ResultMixinStr(builtins.object)
    DefragResult(DefragResult, _ResultMixinStr)

class DefragResult(DefragResult, _ResultMixinStr)
 |  DefragResult(url, fragment)
 |
 |  A 2-tuple that contains the url without fragment identifier and the fragment
 |  identifier as a separate argument.
 |
 |  Method resolution order:
 |      DefragResult
 |      DefragResult
 |      builtins.tuple
 |      _ResultMixinStr
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  geturl(self)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  _encoded_counterpart = <class 'urllib.parse.DefragResultBytes'>
 |      DefragResult(url, fragment)
 |
 |      A 2-tuple that contains the url without fragment identifier and the fragment
 |      identifier as a separate argument.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from DefragResult:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new DefragResult object replacing specified fields with new values
 |
```

```
 |  ----------------------------------------------------------------------
 |  Class methods inherited from DefragResult:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at 0x000000005CDCC0D0>, le
 |      Make a new DefragResult object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from DefragResult:
 |
 |  __new__(_cls, url, fragment)
 |      Create new instance of DefragResult(url, fragment)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from DefragResult:
 |
 |  url
 |      The URL with no fragment identifier.
 |
 |  fragment
 |      Fragment identifier separated from URL, that allows indirect identification of a
 |      secondary resource by reference to a primary resource and additional identifying
 |      information.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from DefragResult:
 |
 |  _fields = ('url', 'fragment')
 |
 |  _source = "from builtins import property as _property, tupl..._itemget...
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
```

```
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.n
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  count(...)
 |      T.count(value) -> integer -- return number of occurrences of value
 |
 |  index(...)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _ResultMixinStr:
 |
 |  encode(self, encoding='ascii', errors='strict')

class DefragResultBytes(DefragResult, _ResultMixinBytes)
 |  DefragResult(url, fragment)
 |
 |  A 2-tuple that contains the url without fragment identifier and the fragment
 |  identifier as a separate argument.
 |
```

```
 |  Method resolution order:
 |      DefragResultBytes
 |      DefragResult
 |      builtins.tuple
 |      _ResultMixinBytes
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  geturl(self)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  _decoded_counterpart = <class 'urllib.parse.DefragResult'>
 |      DefragResult(url, fragment)
 |
 |      A 2-tuple that contains the url without fragment identifier and the fragment
 |      identifier as a separate argument.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from DefragResult:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new DefragResult object replacing specified fields with new values
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from DefragResult:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at 0x000000005CDCC0D0>, len
 |      Make a new DefragResult object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from DefragResult:
 |
 |  __new__(_cls, url, fragment)
 |      Create new instance of DefragResult(url, fragment)
 |
 |  ----------------------------------------------------------------------
```

```
 |  Data descriptors inherited from DefragResult:
 |
 |  url
 |      The URL with no fragment identifier.
 |
 |  fragment
 |      Fragment identifier separated from URL, that allows indirect identification of a
 |      secondary resource by reference to a primary resource and additional identifying
 |      information.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from DefragResult:
 |
 |  _fields = ('url', 'fragment')
 |
 |  _source = "from builtins import property as _property, tupl..._itemget...
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
```

```
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.n
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  count(...)
 |      T.count(value) -> integer -- return number of occurrences of value
 |
 |  index(...)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _ResultMixinBytes:
 |
 |  decode(self, encoding='ascii', errors='strict')

class ParseResult(ParseResult, _NetlocResultMixinStr)
 |  ParseResult(scheme, netloc, path, params,  query, fragment)
 |
 |  A 6-tuple that contains components of a parsed URL.
 |
 |  Method resolution order:
 |      ParseResult
 |      ParseResult
 |      builtins.tuple
 |      _NetlocResultMixinStr
 |      _NetlocResultMixinBase
 |      _ResultMixinStr
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  geturl(self)
 |
 |  ----------------------------------------------------------------------
```

```
|  Data and other attributes defined here:
|
|  _encoded_counterpart = <class 'urllib.parse.ParseResultBytes'>
|      ParseResult(scheme, netloc, path, params,  query, fragment)
|
|      A 6-tuple that contains components of a parsed URL.
|
|  ----------------------------------------------------------------------
|  Methods inherited from ParseResult:
|
|  __getnewargs__(self)
|      Return self as a plain tuple.  Used by copy and pickle.
|
|  __repr__(self)
|      Return a nicely formatted representation string
|
|  _asdict(self)
|      Return a new OrderedDict which maps field names to their values.
|
|  _replace(_self, **kwds)
|      Return a new ParseResult object replacing specified fields with new values
|
|  ----------------------------------------------------------------------
|  Class methods inherited from ParseResult:
|
|  _make(iterable, new=<built-in method __new__ of type object at 0x000000005CDCC0D0>, len
|      Make a new ParseResult object from a sequence or iterable
|
|  ----------------------------------------------------------------------
|  Static methods inherited from ParseResult:
|
|  __new__(_cls, scheme, netloc, path, params, query, fragment)
|      Create new instance of ParseResult(scheme, netloc, path, params, query, fragment)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from ParseResult:
|
|  scheme
|      Specifies URL scheme for the request.
|
|  netloc
|      Network location where the request is made to.
|
|  path
|      The hierarchical path, such as the path to a file to download.
|
|  params
|      Parameters for last path element used to dereference the URI in order to provide
```

```
 |      access to perform some operation on the resource.
 |
 |  query
 |      The query component, that contains non-hierarchical data, that along with data
 |      in path component, identifies a resource in the scope of URI's scheme and
 |      network location.
 |
 |  fragment
 |      Fragment identifier, that allows indirect identification of a secondary resource
 |      by reference to a primary resource and additional identifying information.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from ParseResult:
 |
 |  _fields = ('scheme', 'netloc', 'path', 'params', 'query', 'fragment')
 |
 |  _source = "from builtins import property as _property, tupl..._itemget...
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
```

```
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.n
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  count(...)
 |      T.count(value) -> integer -- return number of occurrences of value
 |
 |  index(...)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _NetlocResultMixinStr:
 |
 |  _hostinfo
 |
 |  _userinfo
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _NetlocResultMixinBase:
 |
 |  hostname
 |
 |  password
 |
 |  port
 |
 |  username
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _ResultMixinStr:
 |
 |  encode(self, encoding='ascii', errors='strict')
```

```
class ParseResultBytes(ParseResult, _NetlocResultMixinBytes)
 |  ParseResult(scheme, netloc, path, params,  query, fragment)
 |
 |  A 6-tuple that contains components of a parsed URL.
 |
 |  Method resolution order:
 |      ParseResultBytes
 |      ParseResult
 |      builtins.tuple
 |      _NetlocResultMixinBytes
 |      _NetlocResultMixinBase
 |      _ResultMixinBytes
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  geturl(self)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  _decoded_counterpart = <class 'urllib.parse.ParseResult'>
 |      ParseResult(scheme, netloc, path, params,  query, fragment)
 |
 |      A 6-tuple that contains components of a parsed URL.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from ParseResult:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new ParseResult object replacing specified fields with new values
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from ParseResult:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at 0x000000005CDCC0D0>, len
 |      Make a new ParseResult object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
```

```
 |  Static methods inherited from ParseResult:
 |
 |  __new__(_cls, scheme, netloc, path, params, query, fragment)
 |      Create new instance of ParseResult(scheme, netloc, path, params, query, fragment)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from ParseResult:
 |
 |  scheme
 |      Specifies URL scheme for the request.
 |
 |  netloc
 |      Network location where the request is made to.
 |
 |  path
 |      The hierarchical path, such as the path to a file to download.
 |
 |  params
 |      Parameters for last path element used to dereference the URI in order to provide
 |      access to perform some operation on the resource.
 |
 |  query
 |      The query component, that contains non-hierarchical data, that along with data
 |      in path component, identifies a resource in the scope of URI's scheme and
 |      network location.
 |
 |  fragment
 |      Fragment identifier, that allows indirect identification of a secondary resource
 |      by reference to a primary resource and additional identifying information.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from ParseResult:
 |
 |  _fields = ('scheme', 'netloc', 'path', 'params', 'query', 'fragment')
 |
 |  _source = "from builtins import property as _property, tupl..._itemget...
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
```

```
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  count(...)
|      T.count(value) -> integer -- return number of occurrences of value
|
|  index(...)
|      T.index(value, [start, [stop]]) -> integer -- return first index of value.
|      Raises ValueError if the value is not present.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from _NetlocResultMixinBytes:
|
|  _hostinfo
```

```
 |
 |  _userinfo
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _NetlocResultMixinBase:
 |
 |  hostname
 |
 |  password
 |
 |  port
 |
 |  username
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _ResultMixinBytes:
 |
 |  decode(self, encoding='ascii', errors='strict')

class SplitResult(SplitResult, _NetlocResultMixinStr)
 |  SplitResult(scheme, netloc, path, query, fragment)
 |
 |  A 5-tuple that contains the different components of a URL. Similar to
 |  ParseResult, but does not split params.
 |
 |  Method resolution order:
 |      SplitResult
 |      SplitResult
 |      builtins.tuple
 |      _NetlocResultMixinStr
 |      _NetlocResultMixinBase
 |      _ResultMixinStr
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  geturl(self)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  _encoded_counterpart = <class 'urllib.parse.SplitResultBytes'>
 |      SplitResult(scheme, netloc, path, query, fragment)
 |
 |      A 5-tuple that contains the different components of a URL. Similar to
 |      ParseResult, but does not split params.
 |
 |  ----------------------------------------------------------------------
```

```
 |  Methods inherited from SplitResult:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new SplitResult object replacing specified fields with new values
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from SplitResult:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at 0x000000005CDCC0D0>, le
 |      Make a new SplitResult object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from SplitResult:
 |
 |  __new__(_cls, scheme, netloc, path, query, fragment)
 |      Create new instance of SplitResult(scheme, netloc, path, query, fragment)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from SplitResult:
 |
 |  scheme
 |      Specifies URL scheme for the request.
 |
 |  netloc
 |      Network location where the request is made to.
 |
 |  path
 |      The hierarchical path, such as the path to a file to download.
 |
 |  query
 |      The query component, that contains non-hierarchical data, that along with data
 |      in path component, identifies a resource in the scope of URI's scheme and
 |      network location.
 |
 |  fragment
 |      Fragment identifier, that allows indirect identification of a secondary resource
 |      by reference to a primary resource and additional identifying information.
 |
 |  ----------------------------------------------------------------------
```

```
|  Data and other attributes inherited from SplitResult:
|
|  _fields = ('scheme', 'netloc', 'path', 'query', 'fragment')
|
|  _source = "from builtins import property as _property, tupl..._itemget...
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.tuple:
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __contains__(self, key, /)
|      Return key in self.
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
```

```
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __rmul__(self, value, /)
 |      Return self*value.
 |
 |  count(...)
 |      T.count(value) -> integer -- return number of occurrences of value
 |
 |  index(...)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _NetlocResultMixinStr:
 |
 |  _hostinfo
 |
 |  _userinfo
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _NetlocResultMixinBase:
 |
 |  hostname
 |
 |  password
 |
 |  port
 |
 |  username
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _ResultMixinStr:
 |
 |  encode(self, encoding='ascii', errors='strict')

class SplitResultBytes(SplitResult, _NetlocResultMixinBytes)
 |  SplitResult(scheme, netloc, path, query, fragment)
 |
 |  A 5-tuple that contains the different components of a URL. Similar to
 |  ParseResult, but does not split params.
 |
 |  Method resolution order:
 |      SplitResultBytes
 |      SplitResult
 |      builtins.tuple
 |      _NetlocResultMixinBytes
 |      _NetlocResultMixinBase
```

```
 |      _ResultMixinBytes
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  geturl(self)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  _decoded_counterpart = <class 'urllib.parse.SplitResult'>
 |      SplitResult(scheme, netloc, path, query, fragment)
 |
 |      A 5-tuple that contains the different components of a URL. Similar to
 |      ParseResult, but does not split params.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from SplitResult:
 |
 |  __getnewargs__(self)
 |      Return self as a plain tuple.  Used by copy and pickle.
 |
 |  __repr__(self)
 |      Return a nicely formatted representation string
 |
 |  _asdict(self)
 |      Return a new OrderedDict which maps field names to their values.
 |
 |  _replace(_self, **kwds)
 |      Return a new SplitResult object replacing specified fields with new values
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from SplitResult:
 |
 |  _make(iterable, new=<built-in method __new__ of type object at 0x000000005CDCC0D0>, len
 |      Make a new SplitResult object from a sequence or iterable
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from SplitResult:
 |
 |  __new__(_cls, scheme, netloc, path, query, fragment)
 |      Create new instance of SplitResult(scheme, netloc, path, query, fragment)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from SplitResult:
 |
 |  scheme
 |      Specifies URL scheme for the request.
```

```
 |
 |  netloc
 |      Network location where the request is made to.
 |
 |  path
 |      The hierarchical path, such as the path to a file to download.
 |
 |  query
 |      The query component, that contains non-hierarchical data, that along with data
 |      in path component, identifies a resource in the scope of URI's scheme and
 |      network location.
 |
 |  fragment
 |      Fragment identifier, that allows indirect identification of a secondary resource
 |      by reference to a primary resource and additional identifying information.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from SplitResult:
 |
 |  _fields = ('scheme', 'netloc', 'path', 'query', 'fragment')
 |
 |  _source = "from builtins import property as _property, tupl..._itemget...
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.tuple:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
```

```
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  count(...)
|      T.count(value) -> integer -- return number of occurrences of value
|
|  index(...)
|      T.index(value, [start, [stop]]) -> integer -- return first index of value.
|      Raises ValueError if the value is not present.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from _NetlocResultMixinBytes:
|
|  _hostinfo
|
|  _userinfo
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from _NetlocResultMixinBase:
|
|  hostname
|
|  password
|
|  port
|
|  username
|
```

```
     |  ----------------------------------------------------------------------
     |  Methods inherited from _ResultMixinBytes:
     |
     |  decode(self, encoding='ascii', errors='strict')

FUNCTIONS
    parse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='repla
        Parse a query given as a string argument.

        Arguments:

        qs: percent-encoded query string to be parsed

        keep_blank_values: flag indicating whether blank values in
            percent-encoded queries should be treated as blank strings.
            A true value indicates that blanks should be retained as
            blank strings.  The default false value indicates that
            blank values are to be ignored and treated as if they were
            not included.

        strict_parsing: flag indicating what to do with parsing errors.
            If false (the default), errors are silently ignored.
            If true, errors raise a ValueError exception.

        encoding and errors: specify how to decode percent-encoded sequences
            into Unicode characters, as accepted by the bytes.decode() method.

        Returns a dictionary.

    parse_qsl(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='repl
        Parse a query given as a string argument.

        Arguments:

        qs: percent-encoded query string to be parsed

        keep_blank_values: flag indicating whether blank values in
            percent-encoded queries should be treated as blank strings.
            A true value indicates that blanks should be retained as blank
            strings.  The default false value indicates that blank values
            are to be ignored and treated as if they were  not included.

        strict_parsing: flag indicating what to do with parsing errors. If
            false (the default), errors are silently ignored. If true,
            errors raise a ValueError exception.

        encoding and errors: specify how to decode percent-encoded sequences
            into Unicode characters, as accepted by the bytes.decode() method.
```

```
    Returns a list, as G-d intended.

quote(string, safe='/', encoding=None, errors=None)
    quote('abc def') -> 'abc%20def'

    Each part of a URL, e.g. the path info, the query, etc., has a
    different set of reserved characters that must be quoted.

    RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax lists
    the following reserved characters.

    reserved    = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" |
                  "$" | ","

    Each of these characters is reserved in some component of a URL,
    but not necessarily in all of them.

    By default, the quote function is intended for quoting the path
    section of a URL.  Thus, it will not encode '/'.  This character
    is reserved, but in typical usage the quote function is being
    called on a path where the existing slash characters are used as
    reserved characters.

    string and safe may be either str or bytes objects. encoding and errors
    must not be specified if string is a bytes object.

    The optional encoding and errors parameters specify how to deal with
    non-ASCII characters, as accepted by the str.encode method.
    By default, encoding='utf-8' (characters are encoded with UTF-8), and
    errors='strict' (unsupported characters raise a UnicodeEncodeError).

quote_from_bytes(bs, safe='/')
    Like quote(), but accepts a bytes object rather than a str, and does
    not perform string-to-bytes encoding.  It always returns an ASCII string.
    quote_from_bytes(b'abc def?') -> 'abc%20def%3f'

quote_plus(string, safe='', encoding=None, errors=None)
    Like quote(), but also replace ' ' with '+', as required for quoting
    HTML form values. Plus signs in the original string are escaped unless
    they are included in safe. It also does not have safe default to '/'.

unquote(string, encoding='utf-8', errors='replace')
    Replace %xx escapes by their single-character equivalent. The optional
    encoding and errors parameters specify how to decode percent-encoded
    sequences into Unicode characters, as accepted by the bytes.decode()
    method.
    By default, percent-encoded sequences are decoded with UTF-8, and invalid
```

```
    sequences are replaced by a placeholder character.

    unquote('abc%20def') -> 'abc def'.

unquote_plus(string, encoding='utf-8', errors='replace')
    Like unquote(), but also replace plus signs by spaces, as required for
    unquoting HTML form values.

    unquote_plus('%7e/abc+def') -> '~/abc def'

unquote_to_bytes(string)
    unquote_to_bytes('abc%20def') -> b'abc def'.

urldefrag(url)
    Removes any existing fragment from URL.

    Returns a tuple of the defragmented URL and the fragment.  If
    the URL contained no fragments, the second element is the
    empty string.

urlencode(query, doseq=False, safe='', encoding=None, errors=None, quote_via=<function quo
    Encode a dict or sequence of two-element tuples into a URL query string.

    If any values in the query arg are sequences and doseq is true, each
    sequence element is converted to a separate parameter.

    If the query arg is a sequence of two-element tuples, the order of the
    parameters in the output will match the order of parameters in the
    input.

    The components of a query arg may each be either a string or a bytes type.

    The safe, encoding, and errors parameters are passed down to the function
    specified by quote_via (encoding and errors only if a component is a str).

urljoin(base, url, allow_fragments=True)
    Join a base URL and a possibly relative URL to form an absolute
    interpretation of the latter.

urlparse(url, scheme='', allow_fragments=True)
    Parse a URL into 6 components:
    <scheme>://<netloc>/<path>;<params>?<query>#<fragment>
    Return a 6-tuple: (scheme, netloc, path, params, query, fragment).
    Note that we don't break the components up in smaller bits
    (e.g. netloc is a single string) and we don't expand % escapes.

urlsplit(url, scheme='', allow_fragments=True)
    Parse a URL into 5 components:
```

```
        <scheme>://<netloc>/<path>?<query>#<fragment>
        Return a 5-tuple: (scheme, netloc, path, query, fragment).
        Note that we don't break the components up in smaller bits
        (e.g. netloc is a single string) and we don't expand % escapes.

    urlunparse(components)
        Put a parsed URL back together again.  This may result in a
        slightly different, but equivalent URL, if the URL that was parsed
        originally had redundant delimiters, e.g. a ? with an empty query
        (the draft states that these are equivalent).

    urlunsplit(components)
        Combine the elements of a tuple as returned by urlsplit() into a
        complete URL as a string. The data argument can be any five-item iterable.
        This may result in a slightly different, but equivalent URL, if the URL that
        was parsed originally had unnecessary delimiters (for example, a ? with an
        empty query; the RFC states that these are equivalent).

DATA
    __all__ = ['urlparse', 'urlunparse', 'urljoin', 'urldefrag', 'urlsplit...

FILE
    c:\users\jtownson\appdata\local\continuum\anaconda3\lib\urllib\parse.py
```

Writing Packages Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which **MUST** contain a special file called `__init__.py`. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called `foo`, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the `__init__.py` file inside the `foo` directory.

To use the module bar, we can import it in two ways:

```
In [12]: #import foo.bar
         ## or
         #from foo import bar
```

In the first method, we must use the `foo` prefix whenever we access the module `bar`. In the second method, we don't, because we import the module to our module's namespace.

The `__init__.py` file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the `__all__` variable, like so:

```
In [14]: #__init__.py:


         #__all__ = ["bar"]
```

Exercise Whoo, that was a doozy. But we made it! Now we just need to solve this problem:

In this exercise, you will need to print an alphabetically sorted list of all functions in the re module, which contain the word find. The solution will be in the second code cell below.

```
In [15]: import re

         # Your code goes here

In [16]: import re

         # Your code goes here
         find_members = []
         for member in dir(re):
             if "find" in member:
                 find_members.append(member)

         print(sorted(find_members))

['findall', 'finditer']
```

1.2.13 Concluding the Basics

And with that we made it all the way through the basic tutorials from the learnpython.org website. We've covered quite a bit! Before heading any further deep into the world of Python coding, I strongly recommend going back and checking anything you were hesitant on. I know that helped me out greatly.

Some may find this to be enough, and that's great! You made it! But for me, I will be diving deeper and checking out the next section about Data Science. Feel free to follow along and I'll continue writing as I have!

1.3 Data Science Tutorials

1.3.1 Numpy Arrays

Numpy arrays are an alternative to Python Lists. In general, these arrays are faster, and easier to work with in general than generic python lists. One of the key features is that they give the user the ability to perform calculations across the entirety of the arrays.

In the below example, we create two Python lists. Then after importing the Numpy package, we create Numpy arrays out of the lists we made.

```
In [17]: # Create 2 new lists height and weight
         height = [1.87,  1.87, 1.82, 1.91, 1.90, 1.85]
         weight = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

         # Import the numpy package as np
         import numpy as np

         # Create 2 numpy arrays from height and weight
```

```
        np_height = np.array(height)
        np_weight = np.array(weight)

        print(type(np_height))
        print(np_height)
```

```
<class 'numpy.ndarray'>
[1.87 1.87 1.82 1.91 1.9  1.85]
```

Notice that the type of the object may have changed from a list to a numpy.ndarray, but when printing the array out itself it doesn't look any different. All of the changes have been done in the background to make it simpler for the user.

Element-wise Calculations Now that we have our height and weight arrays, we can perform element-wise calculations on them. For example, unlike the complexity of lists, using our Numpy arrays we can take all 6 of the height and weight observations above and calculate the BMI for each observation with a single equation. This operation will be much quicker than if we used lists, and more computationally efficient. This efficiency is even more handy when we have 1000s or more observations in our data.

```
In [25]: # Calculate bmi
         bmi = np_weight / (np_height ** 2)

         # Print the result
         print(bmi)
```

```
[23.34925219 27.88755755 28.75558507 25.48723993 23.87257618 25.84368152]
```

Subsetting Another great feature of Numpy arrays is the ability to subset. For instance, if you wanted to know which observations in our BMI array are above 25, we could quickly subset it to find out.

```
In [32]: # For a boolean response
         bmi > 25

         # Print only those observations above 23
         print(bmi[bmi > 25])

         # here's what the Boolean response looks like:
         print(bmi > 25)
```

```
[27.88755755 28.75558507 25.48723993 25.84368152]
[False  True  True  True False  True]
```

Exercise First, convert the list of weights from a list to a Numpy array. Then, convert all of the weights from kilograms to pounds. Use the scalar conversion of 2.2 lbs per kilogram to make your conversion. Lastly, print the resulting array of weights in pounds. The solution is in the second cell below.

```
In [33]: weight_kg = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

         import numpy as np

         # Create a numpy array np_weight_kg from weight_kg


         # Create np_weight_lbs from np_weight_kg

         # Print out np_weight_lbs
```

```
In [35]: weight_kg = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

         import numpy as np

         # Create a numpy array np_weight_kg from weight_kg

         np_weight_kg = np.array(weight_kg)

         # Create np_weight_lbs from np_weight_kg

         np_weight_lbs = np_weight_kg * 2.2

         # Print out np_weight_lbs

         print(np_weight_lbs)

[179.63  214.544 209.55  204.556 189.596 194.59 ]
```

1.3.2 Pandas Basics

This section will be very important, as Pandas is what any Python user in data science crutches on. Like `plyr` and `dplyr` in R, Pandas lets Python users easily get and clean data, among other things. Pandas is a high-level data manipulation tool developed by Wes McKinney. It is built on the Numpy package and its key data structure is called the DataFrame.

Pandas DataFrames DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables (just like dataframes in R but less cool). There are several ways to create a DataFrame. One way way is to use a dictionary. For example:

```
In [2]: dict = {"country": ["Brazil", "Russia", "India", "China", "South Africa"],
                "capital": ["Brasilia", "Moscow", "New Dehli", "Beijing", "Pretoria"],
```

```
             "area": [8.516, 17.10, 3.286, 9.597, 1.221],
             "population": [200.4, 143.5, 1252, 1357, 52.98] }

        import pandas as pd
        brics = pd.DataFrame(dict)
        print(brics)
```

```
        country    capital    area  population
0        Brazil   Brasilia   8.516      200.40
1        Russia     Moscow  17.100      143.50
2         India  New Dehli   3.286     1252.00
3         China    Beijing   9.597     1357.00
4  South Africa   Pretoria   1.221       52.98
```

As you can see with the new brics DataFrame, Pandas has assigned a key for each country as the numerical values 0 through 4. If you would like to have different index values, say, the two letter country code, you can do that easily as well.

```
In [3]: # Set the index for brics
        brics.index = ["BR", "RU", "IN", "CH", "SA"]

        # Print out brics with new index values
        print(brics)
```

```
         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Dehli   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98
```

Another way to create a DataFrame is by importing a csv file using Pandas. If you cloned this repository from my GitHub repo, then in this same directory is a csv file called `cars.csv`. We will import this into our environment by using `pd.read_csv`.

```
In [4]: # Import pandas as pd
        import pandas as pd

        # Import the cars.csv data: cars
        cars = pd.read_csv('cars.csv')

        # Print out cars
        print(cars)
```

```
    YEAR        Make                           Model            Size  \
0   2012  MITSUBISHI                          i-MiEV      SUBCOMPACT
1   2012      NISSAN                            LEAF        MID-SIZE
```

2	2013	FORD	FOCUS ELECTRIC	COMPACT
3	2013	MITSUBISHI	i-MiEV	SUBCOMPACT
4	2013	NISSAN	LEAF	MID-SIZE
5	2013	SMART	FORTWO ELECTRIC DRIVE CABRIOLET	TWO-SEATER
6	2013	SMART	FORTWO ELECTRIC DRIVE COUPE	TWO-SEATER
7	2013	TESLA	MODEL S (40 kWh battery)	FULL-SIZE
8	2013	TESLA	MODEL S (60 kWh battery)	FULL-SIZE
9	2013	TESLA	MODEL S (85 kWh battery)	FULL-SIZE
10	2013	TESLA	MODEL S PERFORMANCE	FULL-SIZE
11	2014	CHEVROLET	SPARK EV	SUBCOMPACT
12	2014	FORD	FOCUS ELECTRIC	COMPACT
13	2014	MITSUBISHI	i-MiEV	SUBCOMPACT
14	2014	NISSAN	LEAF	MID-SIZE
15	2014	SMART	FORTWO ELECTRIC DRIVE CABRIOLET	TWO-SEATER
16	2014	SMART	FORTWO ELECTRIC DRIVE COUPE	TWO-SEATER
17	2014	TESLA	MODEL S (60 kWh battery)	FULL-SIZE
18	2014	TESLA	MODEL S (85 kWh battery)	FULL-SIZE
19	2014	TESLA	MODEL S PERFORMANCE	FULL-SIZE
20	2015	BMW	i3	SUBCOMPACT
21	2015	CHEVROLET	SPARK EV	SUBCOMPACT
22	2015	FORD	FOCUS ELECTRIC	COMPACT
23	2015	KIA	SOUL EV	STATION WAGON - SMALL
24	2015	MITSUBISHI	i-MiEV	SUBCOMPACT
25	2015	NISSAN	LEAF	MID-SIZE
26	2015	SMART	FORTWO ELECTRIC DRIVE CABRIOLET	TWO-SEATER
27	2015	SMART	FORTWO ELECTRIC DRIVE COUPE	TWO-SEATER
28	2015	TESLA	MODEL S (60 kWh battery)	FULL-SIZE
29	2015	TESLA	MODEL S (70 kWh battery)	FULL-SIZE
30	2015	TESLA	MODEL S (85/90 kWh battery)	FULL-SIZE
31	2015	TESLA	MODEL S 70D	FULL-SIZE
32	2015	TESLA	MODEL S 85D/90D	FULL-SIZE
33	2015	TESLA	MODEL S P85D/P90D	FULL-SIZE
34	2016	BMW	i3	SUBCOMPACT
35	2016	CHEVROLET	SPARK EV	SUBCOMPACT
36	2016	FORD	FOCUS ELECTRIC	COMPACT
37	2016	KIA	SOUL EV	STATION WAGON - SMALL
38	2016	MITSUBISHI	i-MiEV	SUBCOMPACT
39	2016	NISSAN	LEAF (24 kWh battery)	MID-SIZE
40	2016	NISSAN	LEAF (30 kWh battery)	MID-SIZE
41	2016	SMART	FORTWO ELECTRIC DRIVE CABRIOLET	TWO-SEATER
42	2016	SMART	FORTWO ELECTRIC DRIVE COUPE	TWO-SEATER
43	2016	TESLA	MODEL S (60 kWh battery)	FULL-SIZE
44	2016	TESLA	MODEL S (70 kWh battery)	FULL-SIZE
45	2016	TESLA	MODEL S (85/90 kWh battery)	FULL-SIZE
46	2016	TESLA	MODEL S 70D	FULL-SIZE
47	2016	TESLA	MODEL S 85D/90D	FULL-SIZE
48	2016	TESLA	MODEL S 90D (Refresh)	FULL-SIZE
49	2016	TESLA	MODEL S P85D/P90D	FULL-SIZE

```
50  2016       TESLA          MODEL S P90D (Refresh)             FULL-SIZE
51  2016       TESLA                    MODEL X 90D         SUV - STANDARD
52  2016       TESLA                   MODEL X P90D         SUV - STANDARD

    (kW) Unnamed: 5 TYPE  CITY (kWh/100 km)  HWY (kWh/100 km)  \
0     49          A1    B               16.9              21.4
1     80          A1    B               19.3              23.0
2    107          A1    B               19.0              21.1
3     49          A1    B               16.9              21.4
4     80          A1    B               19.3              23.0
5     35          A1    B               17.2              22.5
6     35          A1    B               17.2              22.5
7    270          A1    B               22.4              21.9
8    270          A1    B               22.2              21.7
9    270          A1    B               23.8              23.2
10   310          A1    B               23.9              23.2
11   104          A1    B               16.0              19.6
12   107          A1    B               19.0              21.1
13    49          A1    B               16.9              21.4
14    80          A1    B               16.5              20.8
15    35          A1    B               17.2              22.5
16    35          A1    B               17.2              22.5
17   225          A1    B               22.2              21.7
18   270          A1    B               23.8              23.2
19   310          A1    B               23.9              23.2
20   125          A1    B               15.2              18.8
21   104          A1    B               16.0              19.6
22   107          A1    B               19.0              21.1
23    81          A1    B               17.5              22.7
24    49          A1    B               16.9              21.4
25    80          A1    B               16.5              20.8
26    35          A1    B               17.2              22.5
27    35          A1    B               17.2              22.5
28   283          A1    B               22.2              21.7
29   283          A1    B               23.8              23.2
30   283          A1    B               23.8              23.2
31   280          A1    B               20.8              20.6
32   280          A1    B               22.0              19.8
33   515          A1    B               23.4              21.5
34   125          A1    B               15.2              18.8
35   104          A1    B               16.0              19.6
36   107          A1    B               19.0              21.1
37    81          A1    B               17.5              22.7
38    49          A1    B               16.9              21.4
39    80          A1    B               16.5              20.8
40    80          A1    B               17.0              20.7
41    35          A1    B               17.2              22.5
42    35          A1    B               17.2              22.5
```

43	283	A1	B	22.2	21.7
44	283	A1	B	23.8	23.2
45	283	A1	B	23.8	23.2
46	386	A1	B	20.8	20.6
47	386	A1	B	22.0	19.8
48	386	A1	B	20.8	19.7
49	568	A1	B	23.4	21.5
50	568	A1	B	22.9	21.0
51	386	A1	B	23.2	22.2
52	568	A1	B	23.6	23.3

	COMB (kWh/100 km)	CITY (Le/100 km)	HWY (Le/100 km)	COMB (Le/100 km) \
0	18.7	1.9	2.4	2.1
1	21.1	2.2	2.6	2.4
2	20.0	2.1	2.4	2.2
3	18.7	1.9	2.4	2.1
4	21.1	2.2	2.6	2.4
5	19.6	1.9	2.5	2.2
6	19.6	1.9	2.5	2.2
7	22.2	2.5	2.5	2.5
8	21.9	2.5	2.4	2.5
9	23.6	2.7	2.6	2.6
10	23.6	2.7	2.6	2.6
11	17.8	1.8	2.2	2.0
12	20.0	2.1	2.4	2.2
13	18.7	1.9	2.4	2.1
14	18.4	1.9	2.3	2.1
15	19.6	1.9	2.5	2.2
16	19.6	1.9	2.5	2.2
17	21.9	2.5	2.4	2.5
18	23.6	2.7	2.6	2.6
19	23.6	2.7	2.6	2.6
20	16.8	1.7	2.1	1.9
21	17.8	1.8	2.2	2.0
22	20.0	2.1	2.4	2.2
23	19.9	2.0	2.6	2.2
24	18.7	1.9	2.4	2.1
25	18.4	1.9	2.3	2.1
26	19.6	1.9	2.5	2.2
27	19.6	1.9	2.5	2.2
28	21.9	2.5	2.4	2.5
29	23.6	2.7	2.6	2.6
30	23.6	2.7	2.6	2.6
31	20.7	2.3	2.3	2.3
32	21.0	2.5	2.2	2.4
33	22.5	2.6	2.4	2.5
34	16.8	1.7	2.1	1.9
35	17.8	1.8	2.2	2.0

36	20.0	2.1	2.4	2.2
37	19.9	2.0	2.6	2.2
38	18.7	1.9	2.4	2.1
39	18.4	1.9	2.3	2.1
40	18.6	1.9	2.3	2.1
41	19.6	1.9	2.5	2.2
42	19.6	1.9	2.5	2.2
43	21.9	2.5	2.4	2.5
44	23.6	2.7	2.6	2.6
45	23.6	2.7	2.6	2.6
46	20.7	2.3	2.3	2.3
47	21.0	2.5	2.2	2.4
48	20.3	2.3	2.2	2.3
49	22.5	2.6	2.4	2.5
50	22.1	2.6	2.4	2.5
51	22.7	2.6	2.5	2.6
52	23.5	2.7	2.6	2.6

	(g/km)	RATING	(km)	TIME (h)
0	0	NaN	100	7
1	0	NaN	117	7
2	0	NaN	122	4
3	0	NaN	100	7
4	0	NaN	117	7
5	0	NaN	109	8
6	0	NaN	109	8
7	0	NaN	224	6
8	0	NaN	335	10
9	0	NaN	426	12
10	0	NaN	426	12
11	0	NaN	131	7
12	0	NaN	122	4
13	0	NaN	100	7
14	0	NaN	135	5
15	0	NaN	109	8
16	0	NaN	109	8
17	0	NaN	335	10
18	0	NaN	426	12
19	0	NaN	426	12
20	0	NaN	130	4
21	0	NaN	131	7
22	0	NaN	122	4
23	0	NaN	149	4
24	0	NaN	100	7
25	0	NaN	135	5
26	0	NaN	109	8
27	0	NaN	109	8
28	0	NaN	335	10

```
29          0      NaN   377           12
30          0      NaN   426           12
31          0      NaN   386           12
32          0      NaN   435           12
33          0      NaN   407           12
34          0     10.0   130            4
35          0     10.0   131            7
36          0     10.0   122            4
37          0     10.0   149            4
38          0     10.0   100            7
39          0     10.0   135            5
40          0     10.0   172            6
41          0     10.0   109            8
42          0     10.0   109            8
43          0     10.0   335           10
44          0     10.0   377           12
45          0     10.0   426           12
46          0     10.0   386           12
47          0     10.0   435           12
48          0     10.0   473           12
49          0     10.0   407           12
50          0     10.0   435           12
51          0     10.0   414           12
52          0     10.0   402           12
```

Indexing Dataframes There are several ways to index a Pandas DataFrame. One of the easiest ways to do this is by using square bracket notation.

In the example below, you can use square brackets to select one column of the `cars` DataFrame. You can either use a single bracket or a double bracket. The single bracket with output a Pandas Series, while a double bracket will output a Pandas DataFrame.

```
In [6]: # Import pandas and cars.csv
        import pandas as pd
        cars = pd.read_csv('cars.csv', index_col = 0)

        # Print out country column as Pandas Series
        print(cars['Model'])

        # Print out country column as Pandas DataFrame
        print(cars[['Model']])

        # Print out DataFrame with country and drives_right columns
        print(cars[['Model', 'Make']])
```

```
YEAR
2012                                i-MiEV
2012                                  LEAF
```

2013	FOCUS ELECTRIC
2013	i-MiEV
2013	LEAF
2013	FORTWO ELECTRIC DRIVE CABRIOLET
2013	FORTWO ELECTRIC DRIVE COUPE
2013	MODEL S (40 kWh battery)
2013	MODEL S (60 kWh battery)
2013	MODEL S (85 kWh battery)
2013	MODEL S PERFORMANCE
2014	SPARK EV
2014	FOCUS ELECTRIC
2014	i-MiEV
2014	LEAF
2014	FORTWO ELECTRIC DRIVE CABRIOLET
2014	FORTWO ELECTRIC DRIVE COUPE
2014	MODEL S (60 kWh battery)
2014	MODEL S (85 kWh battery)
2014	MODEL S PERFORMANCE
2015	i3
2015	SPARK EV
2015	FOCUS ELECTRIC
2015	SOUL EV
2015	i-MiEV
2015	LEAF
2015	FORTWO ELECTRIC DRIVE CABRIOLET
2015	FORTWO ELECTRIC DRIVE COUPE
2015	MODEL S (60 kWh battery)
2015	MODEL S (70 kWh battery)
2015	MODEL S (85/90 kWh battery)
2015	MODEL S 70D
2015	MODEL S 85D/90D
2015	MODEL S P85D/P90D
2016	i3
2016	SPARK EV
2016	FOCUS ELECTRIC
2016	SOUL EV
2016	i-MiEV
2016	LEAF (24 kWh battery)
2016	LEAF (30 kWh battery)
2016	FORTWO ELECTRIC DRIVE CABRIOLET
2016	FORTWO ELECTRIC DRIVE COUPE
2016	MODEL S (60 kWh battery)
2016	MODEL S (70 kWh battery)
2016	MODEL S (85/90 kWh battery)
2016	MODEL S 70D
2016	MODEL S 85D/90D
2016	MODEL S 90D (Refresh)
2016	MODEL S P85D/P90D

```
2016            MODEL S P90D (Refresh)
2016                       MODEL X 90D
2016                      MODEL X P90D
Name: Model, dtype: object
                             Model
YEAR
2012                        i-MiEV
2012                          LEAF
2013                FOCUS ELECTRIC
2013                        i-MiEV
2013                          LEAF
2013  FORTWO ELECTRIC DRIVE CABRIOLET
2013      FORTWO ELECTRIC DRIVE COUPE
2013         MODEL S (40 kWh battery)
2013         MODEL S (60 kWh battery)
2013         MODEL S (85 kWh battery)
2013              MODEL S PERFORMANCE
2014                         SPARK EV
2014                   FOCUS ELECTRIC
2014                           i-MiEV
2014                             LEAF
2014  FORTWO ELECTRIC DRIVE CABRIOLET
2014      FORTWO ELECTRIC DRIVE COUPE
2014         MODEL S (60 kWh battery)
2014         MODEL S (85 kWh battery)
2014              MODEL S PERFORMANCE
2015                               i3
2015                         SPARK EV
2015                   FOCUS ELECTRIC
2015                          SOUL EV
2015                           i-MiEV
2015                             LEAF
2015  FORTWO ELECTRIC DRIVE CABRIOLET
2015      FORTWO ELECTRIC DRIVE COUPE
2015         MODEL S (60 kWh battery)
2015         MODEL S (70 kWh battery)
2015      MODEL S (85/90 kWh battery)
2015                      MODEL S 70D
2015                  MODEL S 85D/90D
2015                MODEL S P85D/P90D
2016                               i3
2016                         SPARK EV
2016                   FOCUS ELECTRIC
2016                          SOUL EV
2016                           i-MiEV
2016            LEAF (24 kWh battery)
2016            LEAF (30 kWh battery)
2016  FORTWO ELECTRIC DRIVE CABRIOLET
```

2016	FORTWO ELECTRIC DRIVE COUPE
2016	MODEL S (60 kWh battery)
2016	MODEL S (70 kWh battery)
2016	MODEL S (85/90 kWh battery)
2016	MODEL S 70D
2016	MODEL S 85D/90D
2016	MODEL S 90D (Refresh)
2016	MODEL S P85D/P90D
2016	MODEL S P90D (Refresh)
2016	MODEL X 90D
2016	MODEL X P90D

	Model	Make
YEAR		
2012	i-MiEV	MITSUBISHI
2012	LEAF	NISSAN
2013	FOCUS ELECTRIC	FORD
2013	i-MiEV	MITSUBISHI
2013	LEAF	NISSAN
2013	FORTWO ELECTRIC DRIVE CABRIOLET	SMART
2013	FORTWO ELECTRIC DRIVE COUPE	SMART
2013	MODEL S (40 kWh battery)	TESLA
2013	MODEL S (60 kWh battery)	TESLA
2013	MODEL S (85 kWh battery)	TESLA
2013	MODEL S PERFORMANCE	TESLA
2014	SPARK EV	CHEVROLET
2014	FOCUS ELECTRIC	FORD
2014	i-MiEV	MITSUBISHI
2014	LEAF	NISSAN
2014	FORTWO ELECTRIC DRIVE CABRIOLET	SMART
2014	FORTWO ELECTRIC DRIVE COUPE	SMART
2014	MODEL S (60 kWh battery)	TESLA
2014	MODEL S (85 kWh battery)	TESLA
2014	MODEL S PERFORMANCE	TESLA
2015	i3	BMW
2015	SPARK EV	CHEVROLET
2015	FOCUS ELECTRIC	FORD
2015	SOUL EV	KIA
2015	i-MiEV	MITSUBISHI
2015	LEAF	NISSAN
2015	FORTWO ELECTRIC DRIVE CABRIOLET	SMART
2015	FORTWO ELECTRIC DRIVE COUPE	SMART
2015	MODEL S (60 kWh battery)	TESLA
2015	MODEL S (70 kWh battery)	TESLA
2015	MODEL S (85/90 kWh battery)	TESLA
2015	MODEL S 70D	TESLA
2015	MODEL S 85D/90D	TESLA
2015	MODEL S P85D/P90D	TESLA
2016	i3	BMW

```
2016                           SPARK EV   CHEVROLET
2016                     FOCUS ELECTRIC        FORD
2016                            SOUL EV         KIA
2016                             i-MiEV  MITSUBISHI
2016               LEAF (24 kWh battery)     NISSAN
2016               LEAF (30 kWh battery)     NISSAN
2016  FORTWO ELECTRIC DRIVE CABRIOLET         SMART
2016      FORTWO ELECTRIC DRIVE COUPE         SMART
2016         MODEL S (60 kWh battery)         TESLA
2016         MODEL S (70 kWh battery)         TESLA
2016      MODEL S (85/90 kWh battery)         TESLA
2016                      MODEL S 70D         TESLA
2016                  MODEL S 85D/90D         TESLA
2016            MODEL S 90D (Refresh)         TESLA
2016                MODEL S P85D/P90D         TESLA
2016           MODEL S P90D (Refresh)         TESLA
2016                      MODEL X 90D         TESLA
2016                     MODEL X P90D         TESLA
```

Square brackets can also be used to access observations (rows) from a DataFrame. For example:

```
In [7]: # Import cars data
        import pandas as pd
        cars = pd.read_csv('cars.csv', index_col = 0)

        # Print out first 4 observations
        print(cars[0:4])

        # Print out fifth, sixth, and seventh observation
        print(cars[4:6])
```

```
            Make           Model        Size  (kW) Unnamed: 5 TYPE  \
YEAR
2012  MITSUBISHI          i-MiEV  SUBCOMPACT    49         A1    B
2012      NISSAN            LEAF    MID-SIZE    80         A1    B
2013        FORD  FOCUS ELECTRIC     COMPACT   107         A1    B
2013  MITSUBISHI          i-MiEV  SUBCOMPACT    49         A1    B

      CITY (kWh/100 km)  HWY (kWh/100 km)  COMB (kWh/100 km)  \
YEAR
2012               16.9              21.4               18.7
2012               19.3              23.0               21.1
2013               19.0              21.1               20.0
2013               16.9              21.4               18.7

      CITY (Le/100 km)  HWY (Le/100 km)  COMB (Le/100 km)  (g/km)  RATING  \
YEAR
```

```
2012              1.9              2.4              2.1       0      NaN
2012              2.2              2.6              2.4       0      NaN
2013              2.1              2.4              2.2       0      NaN
2013              1.9              2.4              2.1       0      NaN

      (km)  TIME (h)
YEAR
2012   100         7
2012   117         7
2013   122         4
2013   100         7
        Make                          Model        Size  (kW) Unnamed: 5  \
YEAR
2013  NISSAN                           LEAF    MID-SIZE    80          A1
2013   SMART  FORTWO ELECTRIC DRIVE CABRIOLET  TWO-SEATER    35          A1

     TYPE  CITY (kWh/100 km)  HWY (kWh/100 km)  COMB (kWh/100 km)  \
YEAR
2013    B               19.3              23.0               21.1
2013    B               17.2              22.5               19.6

      CITY (Le/100 km)  HWY (Le/100 km)  COMB (Le/100 km)  (g/km)  RATING  \
YEAR
2013               2.2              2.6               2.4       0     NaN
2013               1.9              2.5               2.2       0     NaN

      (km)  TIME (h)
YEAR
2013   117         7
2013   109         8
```

Finally, we can also use `loc` and `iloc` to perform just about any data selection operation. `loc` is label-based, which means that you have to specify rows and columns based on their row and column labels. `iloc` is integer index based, so you have to specify rows and columns by their integer index like you did in the previous exercise.

```
In [12]: # Import cars data
         import pandas as pd
         cars = pd.read_csv('cars.csv', index_col = 0)

         # Print out observation for Japan
         print(cars.iloc[2])

         # Print out observations for 2012 and 2013
         print(cars.loc[[2012, 2013]])
```

```
Make                          FORD
Model               FOCUS ELECTRIC
```

```
Size                        COMPACT
(kW)                            107
Unnamed: 5                       A1
TYPE                              B
CITY (kWh/100 km)                19
HWY (kWh/100 km)               21.1
COMB (kWh/100 km)                20
CITY (Le/100 km)                2.1
HWY (Le/100 km)                 2.4
COMB (Le/100 km)                2.2
(g/km)                            0
RATING                          NaN
(km)                            122
TIME (h)                          4
Name: 2013, dtype: object
            Make                           Model        Size  (kW)  \
YEAR
2012  MITSUBISHI                          i-MiEV  SUBCOMPACT    49
2012      NISSAN                            LEAF    MID-SIZE    80
2013        FORD                  FOCUS ELECTRIC     COMPACT   107
2013  MITSUBISHI                          i-MiEV  SUBCOMPACT    49
2013      NISSAN                            LEAF    MID-SIZE    80
2013       SMART  FORTWO ELECTRIC DRIVE CABRIOLET  TWO-SEATER    35
2013       SMART      FORTWO ELECTRIC DRIVE COUPE  TWO-SEATER    35
2013       TESLA         MODEL S (40 kWh battery)   FULL-SIZE   270
2013       TESLA         MODEL S (60 kWh battery)   FULL-SIZE   270
2013       TESLA         MODEL S (85 kWh battery)   FULL-SIZE   270
2013       TESLA              MODEL S PERFORMANCE   FULL-SIZE   310

     Unnamed: 5 TYPE  CITY (kWh/100 km)  HWY (kWh/100 km)  COMB (kWh/100 km)  \
YEAR
2012         A1    B               16.9              21.4               18.7
2012         A1    B               19.3              23.0               21.1
2013         A1    B               19.0              21.1               20.0
2013         A1    B               16.9              21.4               18.7
2013         A1    B               19.3              23.0               21.1
2013         A1    B               17.2              22.5               19.6
2013         A1    B               17.2              22.5               19.6
2013         A1    B               22.4              21.9               22.2
2013         A1    B               22.2              21.7               21.9
2013         A1    B               23.8              23.2               23.6
2013         A1    B               23.9              23.2               23.6

      CITY (Le/100 km)  HWY (Le/100 km)  COMB (Le/100 km)  (g/km)  RATING  \
YEAR
2012               1.9              2.4               2.1       0     NaN
2012               2.2              2.6               2.4       0     NaN
2013               2.1              2.4               2.2       0     NaN
```

```
2013                1.9               2.4                 2.1         0       NaN
2013                2.2               2.6                 2.4         0       NaN
2013                1.9               2.5                 2.2         0       NaN
2013                1.9               2.5                 2.2         0       NaN
2013                2.5               2.5                 2.5         0       NaN
2013                2.5               2.4                 2.5         0       NaN
2013                2.7               2.6                 2.6         0       NaN
2013                2.7               2.6                 2.6         0       NaN

      (km)  TIME (h)
YEAR
2012   100         7
2012   117         7
2013   122         4
2013   100         7
2013   117         7
2013   109         8
2013   109         8
2013   224         6
2013   335        10
2013   426        12
2013   426        12
```

1.4 Bonus Tutorials

Now that the basics are covered, I won't update this as often. Anything I add here may be tougher examples and may seem off the path of what we have covered thus far. So if you're just focused on the basics of Python and learning how to write some simple code, the above work should be enough.

On the flip side, if you're looking for more of a challenge, I'm hoping to find some much more interesting problems and have them saved here as they come up. Enjoy!

1.4.1 Generators

Generators are very easy to implement, but a bit difficult to understand.

Generators are used to create iterators, but with a different approach. Generators are simple functions which return an iterable set of items, one at a time, in a special way.

When an iteration over a set of item starts using the for statement, the generator is run. Once the generator's function code reaches a "yield" statement, the generator yields its execution back to the for loop, returning a new value from the set. The generator function can generate as many values (possibly infinite) as it wants, yielding each one in its turn.

Here is a simple example of a generator function which returns 7 random integers:

```
In [34]: import random


         def lottery():
             # returns 6 numbers between 1 and 40
```

```
        for i in range(6):
            yield random.randint(1, 40)

        # returns a 7th number between 1 and 15
        yield random.randint(1,15)

    for random_number in lottery():
           print("And the next number is... %d!" %(random_number))
```

```
And the next number is... 25!
And the next number is... 14!
And the next number is... 35!
And the next number is... 36!
And the next number is... 23!
And the next number is... 6!
And the next number is... 3!
```

This function decides how to generate the random numbers on its own, and executes the yield statements one at a time, pausing in between to yield execution back to the main for loop.

Exercise Write a generator function which returns the Fibonacci series. They are calculated using the following formula: The first two numbers of the series is always equal to 1, and each consecutive number returned is the sum of the last two numbers. Hint: Can you use only two variables in the generator function? Remember that assignments can be done simultaneously.

Solution will be contained in the 2nd code cell below.

Another hint: The code

```
In [43]: a = 1
         b = 2
         a, b = b, a
         print(a,b)
```

```
2 1
```

will simultaneously switch the values of a and b.

```
In [28]: # fill in this function
         def fib():
             pass # this is a null statement which does
                  # nothing when executed, useful as a placeholder.

         # testing code
         import types
         if type(fib()) == types.GeneratorType:
             print("Good, The fib function is a generator.")

             counter = 0
```

```
    for n in fib():
        print(n)
        counter += 1
        if counter == 10:
            break
```

```
2 1
```

```
In [45]: # fill in this function
         def fib():
             a = 1
             b = 1

             while 1:
                 yield a
                 a, b = b, a + b

         # testing code
         import types
         if type(fib()) == types.GeneratorType:
             print("Good, The fib function is a generator.")

             counter = 0
             for n in fib():
                 print(n)
                 counter += 1
                 if counter == 10:
                     break
```

```
Good, The fib function is a generator.
1
1
2
3
5
8
13
21
34
55
```