01_python

September 20, 2018

1 The Python Programming Language

- **Python is an interpreted language** which means it isn't compiled directly to machine code and importantly is commonly used in an interactive fashion. In Python, you can start the interactive interpreter and begin writing code, line by line, with the interpreter evaluating each statement as you write it. This turns out to be very useful for tasks that require a lot of investigation
- **Python is a dynamically typed language**, similar to languages like JavaScript. This means that when you declare a variable, you can assign it to be an integer on one line, and a string on the next line. Since there is no compilation step, you don't have anyone to help you manage types. You need to either check for the presence of functionality when you go to use it or try and use the functionality and catch any errors that occur.

We can **run this cell by hitting *shift+enter***. What's happening underneath is that the browser is sending your Python code across to a Python interpreter, which executes the code, and sends the results back

```
In [133]: x = 1
          y = 2
          x + y
```

```
Out[133]: 3
```

If we just put in x and execute, we get the value of 1. So it's important to know that the **Python interpreter is stateful**. That is, that your variables exist between cells. Beyond that, if we go back and change something in a previous cell, we have to re-execute the script to make those changes take place. The ***Restart & Run All* option in the *Kernel* menu** is particularly useful, as it wipes the interpreter state and reruns all of the cells in the current notebook.

```
In [2]: x
```

```
Out[2]: 1
```

2 Functions

Python doesn't require the use of keywords like var to declare a variable name or semicolons at the end of lines which are commonly used in other languages. **Python leverages white space to understand the scope of functions and loops and end of line markers to understand the end of statements**. Of course, Python has traditional software structures like functions. Here's an example.

The def statement indicates that we're writing a function. Then each line that is part of the function needs to be indented with a tab character or a couple of spaces. Again, because we're in an interactive environment, when the statement is evaluated on a shift+enter, the results are printed out immediately below

add_numbers is a function that takes two numbers and adds them together.

```
In [3]: def add_numbers(x, y):
            return x + y

        add_numbers(1, 2)

Out[3]: 3
```

Functions are a bit different than you might find in other languages and here are some of subtleties involved.

- First, since there's no typing, **you don't have to set your return type**.
- Second, you don't have to set user return statement at all actually. **There's a special value called None that's returned**. None a similar to null in Java and represents the absence of value.
- Third, in Python, you can have default values for parameters.

Here's an example. In this example, we can rewrite the add numbers function to take three parameters, but we could set the last parameter to be None by default. This means that you can call add numbers with just two values or with three. All of the optional parameters, the ones that you got default values for, need to come at the end of the function declaration.

add_numbers updated to take an optional 3rd parameter. Using print allows printing of multiple expressions within a single cell.

```
In [135]: def add_numbers(x,y,z=None):
              if (z==None):
                  return x+y
              else:
                  return x+y+z

          print(add_numbers(1, 2))

          print(add_numbers(1, 2, 3))

3
6
```

You can also pass optional parameters as labeled values. For example, we can rewrite `add_numbers` so that there's two optional parameters. If we want to call the function with just two numbers but also set the `flag` value, we have to explicitly name and set the flag parameter to `True` when invoking the function.

`add_numbers` updated to take an optional flag parameter.

```
In [5]: def add_numbers(x, y, z=None, flag=False):
            if (flag):
                print('Flag is true!')
            if (z==None):
                return x + y
            else:
                return x + y + z

        print(add_numbers(1, 2, flag=True))
```

```
Flag is true!
3
```

In Python, you can assign a function to a variable. By assigning a function to a variable, you can pass that variable into other functions allowing some basic functional programming. Here's an example where we define a function to add numbers, then we assign that function to a variable, and then we invoke the variable.

Assign function `add_numbers` to variable `a`.

```
In [6]: def add_numbers(x,y):
            return x+y

        a = add_numbers
        a(1,2)
```

```
Out[6]: 3
```

3 Types and Sequences

The absence of static typing in Python doesn't mean that there aren't types. **The Python language has a built in function called `type` which will show you what type a given reference is**. Some of the common types includes strings, the `NoneType`, integers and floating point variables. A function type also exist.

Typed objects have properties associated with them, and these properties can be data or functions.

Use `type` to return the object's type.

```
In [7]: type('This is a string')
```

```
Out[7]: str
```

```
In [8]: type(None)
```

```
Out[8]: NoneType
```

```
In [9]: type(1)
```

```
Out[9]: int
```

```
In [10]: type(1.0)
```

```
Out[10]: float
```

```
In [11]: type(add_numbers)
```

```
Out[11]: function
```

A lot of Python's built around different kinds of sequences or collection types. And **there's three native kinds of collections** that we're going to talk about: - tuples, - lists, and - dictionaries.

A tuple is a sequence of variables which itself is immutable. That means that a tuple has items in an ordering, but that it cannot be changed once created. We **write tuples using parentheses**, and **we can mix types for the contents of the tuple**.

Here's a tuple which has four items. Two are numbers, and two are strings. In Python, either single or double quotes can be used to denote string values.

Tuples are an immutable data structure (cannot be altered).

```
In [12]: x = (1, 'a', 2, 'b')
         type(x)
```

```
Out[12]: tuple
```

Lists are very similar, but they can be mutable, so you can change their length, number of elements, and the element values. A list is declared using the square brackets.

Lists are a mutable data structure.

```
In [13]: x = [1, 'a', 2, 'b']
         type(x)
```

```
Out[13]: list
```

There are a couple of different ways to change the contents of a list. One is through the `append` function which allows you to append new items to the end of the list.

Use `append` **to append an object to a list.**

```
In [14]: x.append(3.3)
         print(x)
```

```
[1, 'a', 2, 'b', 3.3]
```

Both lists and tuples are iterable types, so you can write loops to go through every value they hold. The norm, if you want to look each item in the list is to use a `for` statement. This is similar to the `for each` loop in languages like Java and C# but note that there's no typing required.

This is an example of how to loop through each item in the list.

```
In [15]: for item in x:
             print(item)
```

```
1
a
2
b
3.3
```

Lists and tuples can also be accessed as arrays might in other languages, by using the square bracket operator, which is called the indexing operator. The first item of the list starts at position zero and to get the length of the list, we use the built in `len` function.

For using the indexing operator:

```
In [16]: i=0
         while( i != len(x) ):
             print(x[i])
             i = i + 1
```

```
1
a
2
b
3.3
```

There are some other common functions that you might expect like `min` and `max` which will find the minimum or maximum values in a given list or tuple. Python lists and tuples also have some basic mathematical operations that can be allowed on them. **The plus sign concatenates lists** for instance.

Use + to concatenate lists.

```
In [17]: [1,2] + [3,4]
```

```
Out[17]: [1, 2, 3, 4]
```

And **the asterisks repeats the values of a list**.

Use * to repeat lists.

```
In [18]: [1]*3
```

```
Out[18]: [1, 1, 1]
```

A very common operator is the `in` operator. This looks at set membership and returns a boolean value of true or false depending on whether one item is in a given list.

Use the `in` operator to check if something is inside a list.

```
In [19]: 1 in [1, 2, 3]
```

```
Out[19]: True
```

3.1 Slicing

Perhaps the most interesting operations you can do with lists are called *slicing*. In Python, **the indexing operator allows you to submit multiple values**. The first parameter is the starting location, if this is the only element then one item is return from the list. The second parameter is the end of the slice. It's an exclusive end so if you slice with the first parameter being zero and the next parameter being one, then you only get back one item.

This is much easier to explain with an example. One handy aspect of Python is that all strings are actually just lists of characters so slicing works wonderfully on them.

```
In [20]: x = 'This is a string'
         print(x[0]) #first character
         print(x[0:1]) #first character, but we have explicitly set the end character
         print(x[0:2]) #first two characters
```

```
T
T
Th
```

Our **indexing values can also be negative** which is really cool. And this means to index from the back of the string. So `x[-1]` gets us the last letter of the string,

```
In [21]: x[-1]
```

```
Out[21]: 'g'
```

This will return the slice starting from the 4th element from the end and stopping before the 2nd element from the end.

```
In [22]: x[-4:-2]
```

```
Out[22]: 'ri'
```

Finally if we want to reference the start or the end of the string implicitly, we can by just leaving the parameter empty.

So `x[:3]` starts with the first character and goes until position three.

```
In [23]: x[:3]
```

```
Out[23]: 'Thi'
```

And the x[3:] starts with the fourth character because indexing always begins with zero and goes to the end of the list.

Slicing is core to the Python language and is a big part of the scientific computing with Python as well. Especially if you start manipulating matrices.

```
In [24]: x[3:]
```

```
Out[24]: 's is a string'
```

3.2 String manipulation

As we saw, **strings are just lists of characters**. So operations you can do on a list, you can do on a string. This means that you can concatenate two strings together using the plus operator. And multiplying strings will repeat a given string. You can also search for strings using the `in` operator.

```
In [25]: firstname = 'Christopher'
         lastname = 'Brooks'

         print(firstname + ' ' + lastname)
         print(firstname*3)
         print('Chris' in firstname)
```

```
Christopher Brooks
ChristopherChristopherChristopher
True
```

The string type has an associated function called `split`. This function **breaks the string up into substrings based on a simple pattern**.

Here for instance, I'll just split a name based on the presence of a space character. The result is a list of four elements. We can choose the first element with the indexing operator to be the first name, and the last element to be the last name.

```
In [26]: firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the first el
         lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the last el
         print(firstname)
         print(lastname)
```

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

```
In [137]: 'Chris' + 2
```

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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-137-9d01956b24db> in <module>()
    ----> 1 'Chris' + 2


        TypeError: must be str, not int
```

```
In [30]: 'Chris' + str(2)
```

```
Out[30]: 'Chris2'
```

3.3 Dictionaries

Dictionaries are similar to lists and tuples in that they hold a collection of items, but they're labeled collections which do not have an ordering. This means that for each value you insert into the dictionary, you must also give a key to get that value out. **In other languages the structure is often called a map. And in Python we use curly braces to denote a dictionary**.

Here is an example where we might link names to email addresses. You can see that **we indicate each item of the dictionary when creating it using a pair of values separated by colons**. Then **you can retrieve a value for a given label using the indexing operator**.

The types you use for indices or values in the dictionary can be anything. And this could be a mixture of types if you prefer.

```
In [31]: x = {'Christopher Brooks': 'brooksch@umich.edu',
              'Bill Gates': 'billg@microsoft.com'}
         x['Christopher Brooks'] # Retrieve a value by using the indexing operator

Out[31]: 'brooksch@umich.edu'
```

We can add new items to the dictionary using the same indexing operator we are used to. Just on the left hand side of a statement.

```
In [29]: x['Kevyn Collins-Thompson'] = None
         x['Kevyn Collins-Thompson']
```

You can iterate over all of the items in a dictionary in a number of ways.
First you can **iterate over all of the keys** and just pull the contents out as you see fit

```
In [ ]: for name in x:
            print(x[name])
```

Or you can **iterate over the values** and just ignore the keys.

```
In [ ]: for email in x.values():
            print(email)
```

Finally you can **iterate over both the values and the keys** at once using the `items` function.

```
In [ ]: for name, email in x.items():
            print(name)
            print(email)
```

3.4 Unpacking

In Python **you can have a sequence**, that's a list or a tuple of values, **and you can unpack those items into different variables through assignment in one statement**.

Here's an example of that, where we have a tuple that has a first name, last name, and email address. I declare three variables and assign them to the tuple. Underneath, Python has unpacked the tuple, and assigned each of these variables in order.

```
In [32]: x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
         fname, lname, email = x
```

```
In [33]: fname

Out[33]: 'Christopher'

In [34]: lname

Out[34]: 'Brooks'
```

We can see that if we add a fourth item to the tuple, Python isn't sure how to unpack that, so we have an error.

```
In [35]: x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
         fname, lname, email = x


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-35-d2c50ec4987a> in <module>()
          1 x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
    ----> 2 fname, lname, email = x


        ValueError: too many values to unpack (expected 3)
```

3.5 More on Strings

Python 3 uses Unicode by default so there is no problem in dealing with international character sets.

In addition to Unicode, **Python uses a special language for formatting the output of strings**. One of the challenges with dynamic typing is that it's bit unclear when you have to do type conversion yourself. We saw that if we wanted to print out a name and a number that we can't use concatenation without calling the `str` function to convert the number to a string first. This creates a lot of nasty looking code where **every operator you're looking to concatenate is wrapped in this `str` function**.

```
In [36]: print('Chris' + 2)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-36-928a1e955b60> in <module>()
    ----> 1 print('Chris' + 2)


        TypeError: must be str, not int
```

```
In [37]: print('Chris' + str(2))
```

```
Chris2
```

The Python string formatting mini language allows you to write a string statement indicating placeholders for variables to be evaluated. You then pass these variables in either named or in order arguments, and Python handles the string manipulation for you.

Here's an example. Imagine we have purchase order details in a dictionary, which includes a number of items, a price, and a person's name. We can write a sales statement string which includes these items using curly brackets. We can then call the `format` method on that string and pass in the values that we want substituted as appropriate.

```
In [38]: sales_record = {
         'price': 3.24,
         'num_items': 4,
         'person': 'Chris'}

         sales_statement = '{} bought {} item(s) at a price of {} each for a total of {}'

         print(sales_statement.format(sales_record['person'],
                                      sales_record['num_items'],
                                      sales_record['price'],
                                      sales_record['num_items']*sales_record['price']))
```

```
Chris bought 4 item(s) at a price of 3.24 each for a total of 12.96
```

String manipulation is a big part of data cleaning, and the string formatting language allows you to do much more than this. You can control a number of different things like decimal places, for floating point numbers, or whether you want to prepend the positive numbers with the plus sign, or set the alignment of strings to left or right justified, or even enable to use of scientific notation.

Documentation of the Format Specification Mini-Language

4 Dates and Times

A lot of analysis you do might relate to dates and times. For instance, finding the average number of sales over a given period, selecting a list of products to determine if they were purchased in a given period, or trying to find the period with the most activity in online discussion forum systems.

You should be aware that date and times can be stored in many different ways. **One of the most common legacy methods for storing the date and time in online transactions systems is based on the offset from the *epoch*, which is January 1, 1970 (Unix time)**. There's a lot of historical cruft around this, but it's not uncommon to see systems storing the date of a transaction in seconds or milliseconds since this date. So if you see large numbers where you expect to see date and time, you'll need to convert them to make much sense out of the data.

```
In [49]: import datetime as dt
         import time as tm
```

`time` **returns the current time in seconds since the Epoch. (January 1st, 1970)**

```
In [50]: tm.time()

Out[50]: 1537268468.49
```

You can then create a time stamp using the `fromtimestamp` function on the `datetime` object. When we print this value out, we see that the year, month, day, and so forth are also printed out.

Convert the timestamp to datetime.

```
In [51]: dtnow = dt.datetime.fromtimestamp(tm.time())
         dtnow

Out[51]: datetime.datetime(2018, 9, 18, 13, 1, 11, 749109)
```

Handy datetime attributes:

```
In [52]: dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year,

Out[52]: (2018, 9, 18, 13, 1, 11)
```

`datetime` **objects allow for simple math using time deltas**. For instance, here, we can create a time delta of 100 days, then do subtraction and comparisons with the `datetime` object. This is commonly used in data science for creating sliding windows. For instance, where you might want to look for any five day span of time where sales were highest, and flag that for follow up

`timedelta` is a duration expressing the difference between two dates.

```
In [54]: delta = dt.timedelta(days = 100) # create a timedelta of 100 days
         delta

Out[54]: datetime.timedelta(100)
```

`date.today` returns the current local date.

```
In [58]: today = dt.date.today()

In [56]: today - delta # the date 100 days ago

Out[56]: datetime.date(2018, 6, 10)

In [57]: today > today-delta # compare dates

Out[57]: True
```

5 Objects and map()

You can define a class using a `class` keyword, and ending with a colon. Anything indented below this, is within the scope of the class.

Classes in Python are generally named using camel case, which means the first character of each word is capitalized. **You don't declare variables within the object, you just start using them. Class variables can also be declared. These are just variables which are shared across all instances**. So in this example, we're saying that the default for all people is at the 'School of Information'.

To define a method, you just write it as you would have a function. The one change, is that to have access to the instance which a method is being invoked upon, **you must include `self`, in the method signature**. Similarly, if you want **to refer to instance variables set on the object, you prepend them with the word** `self`, with a full stop.

In this definition of a person, for instance, we have written two methods: `set_name` and `set_location`. And both change instance bound variables, called `name` and `location` respectively.

There's no need for an explicit constructor when creating objects in Python. **You can add a constructor if you want to by declaring the `__init__` method**.

Objects in Python do not have private or protected members. If you instantiate an object, you have full access to any of the methods or attributes of that object.

When we run this cell, we see no output.

```
In [59]: class Person:
             department = 'School of Information' #a class variable

             def __init__ (self, name, location): # constructor
                 self.name = name
                 self.location = location
             def set_name(self, new_name): #a method
                 self.name = new_name
             def set_location(self, new_location):
                 self.location = new_location
```

```
In [61]: person = Person('Pedro González', 'Madrid, Spain')
         message = '{} lives in {} and works in the department {}'
         print(message.format(person.name, person.location, person.department))

         person.set_name('Christopher Brooks')
         person.set_location('Ann Arbor, MI, USA')
         print(message.format(person.name, person.location, person.department))
```

```
Pedro González lives in Madrid, Spain and works in the department School of Information
Christopher Brooks lives in Ann Arbor, MI, USA and works in the department School of Information
```

The map function is one of the basis for functional programming in Python. Functional programming is a programming paradigm in which you explicitly declare all parameters which could change through execution of a given function. Thus functional programming is referred to as being side-effect free, because there is a software contract that describes what can actually change by calling a function. Now, Python isn't a functional programming language in the pure sense. Since you can have many side effects of functions, and certainly you don't have to pass in the parameters of everything that you're interested in changing. But functional programming causes one to think more heavily while chaining operations together. And this really is a sort of underlying theme in much of data science and data cleaning in particular. So, functional programming methods are often used in Python, and it's not uncommon to see a parameter for a function, be a function itself.

The `map` built-in function is one example of a functional programming feature of Python. **The map function signature looks like this: the first parameter is the function that you want executed, and the second parameter, and every following parameter, is something which can be iterated upon. All the iterable arguments are unpacked together, and passed into the given function**. That's a little cryptic, so let's take a look at an example.

Imagine we have two list of numbers, maybe prices from two different stores on exactly the same items. And we wanted to find the minimum that we would have to pay if we bought the cheaper item between the two stores. To do this, we could iterate through each list, comparing items and choosing the cheapest. With map, we can do this comparison in a single statement.

```
In [62]: store1 = [10.00, 11.00, 12.34, 2.34]
         store2 = [9.00, 11.10, 12.34, 2.01]
         cheapest = map(min, store1, store2)
         cheapest

Out[62]: <map at 0x10f06a940>
```

But when we go to print out the map, we see that **we get an odd reference value instead of a list of items** that we're expecting. **This is called lazy evaluation**. In Python, the `map` function returns to you a map object. It doesn't actually try and run the function `min` on two items, until you look inside for a value. This is an interesting design pattern of the language, and it's commonly used when dealing with big data. This allows us to have very efficient memory management, even though something might be computationally complex. **Maps are iterable, just like lists and tuples**, so we can use a `for` loop to look at all of the values in the map.

This passing around of functions and data structures which they should be applied to, is a hallmark of functional programming. It's very common in data analysis and cleaning.

```
In [63]: for item in cheapest:
             print(item)

9.0
11.0
12.34
2.01
```

6 Lambda and List Comprehensions

Lambda's are Python's way of creating anonymous functions. These are the same as other functions, but they have no name. The intent is that they're simple or short lived and it's easier just to write out the function in one line instead of going to the trouble of creating a named function.

The lambda syntax is fairly simple. But it might take a bit of time to get used to. **You declare a lambda function with the word `lambda` followed by a list of arguments, followed by a colon**

and then a single expression. There's only one expression to be evaluated in a lambda. The expression value is returned on execution of the lambda.

The return of a lambda is a function reference. So in this case, you would execute `my_function` and pass in three different parameters. Note that you can't have default values for lambda parameters and you can't have complex logic inside of the lambda itself because you're limited to a single expression. So lambdas are really much more limited than full function definitions.

```
In [64]: my_function = lambda a, b, c : a + b

In [65]: my_function(1, 2, 3)

Out[65]: 3
```

Sequences are structures that we can iterate over, and often we create these through loops or by reading in data from a file. **Python has built in support for creating these collections using a more abbreviated syntax called *list comprehensions***.

Here's an example. First we write up a little for-loop. Here I'm iterating between zero and 1,000 and then checking with the modulus operator to see if the number divided by 2 results in any decimals. If the modulus 2 of the number is zero, then I know it divides evenly so this must be an even number and I'll add it to our list.

```
In [ ]: my_list = []
        for number in range(0, 1000):
            if number % 2 == 0:
                my_list.append(number)
        my_list
```

We can rewrite this as a list comprehension by pulling the iteration on one line. **We start the list comprehension with the value we want in the list. In this case, it's a number. Then we put it in the for-loop, and then finally, we add any condition clauses**. You can see that this is much more compact of a format. And it tends to be faster as well. Just like with lambdas, list comprehensions are a condensed format which may offer readability and performance benefits.

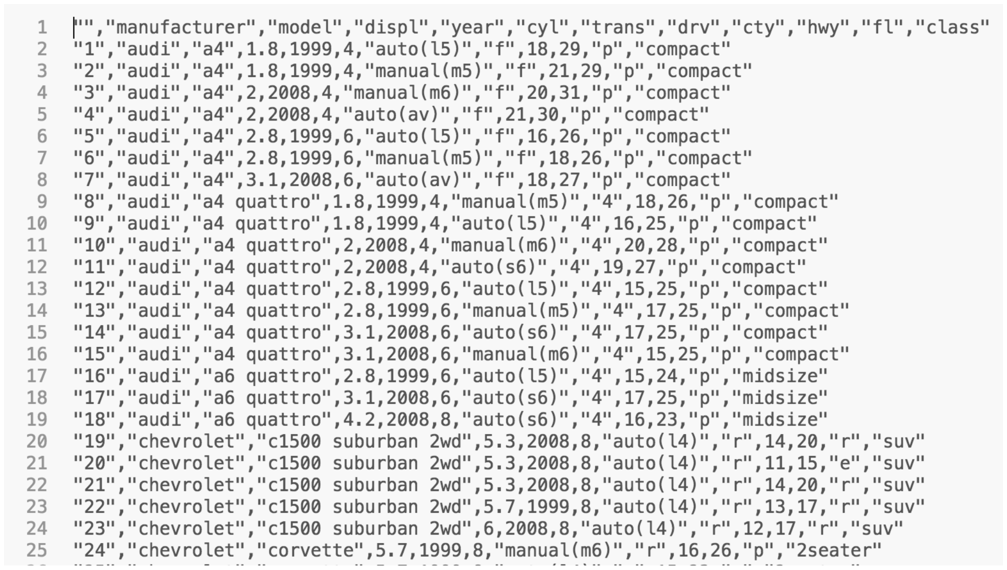
```
In [130]: my_list = [number for number in range(0,1000) if number % 2 == 0]
          my_list[-5:]

Out[130]: [990, 992, 994, 996, 998]
```

7 Reading and Processing CSV files

Let's import our datafile mpg.csv, which contains fuel economy data for 234 cars.

- mpg : miles per gallon
- class : car classification
- cty : city mpg
- cyl : # of cylinders
- displ : engine displacement in liters
- drv : f = front-wheel drive, r = rear wheel drive, 4 = 4wd



```
"","manufacturer","model","displ","year","cyl","trans","drv","cty","hwy","fl","class"
"1","audi","a4",1.8,1999,4,"auto(l5)","f",18,29,"p","compact"
"2","audi","a4",1.8,1999,4,"manual(m5)","f",21,29,"p","compact"
"3","audi","a4",2,2008,4,"manual(m6)","f",20,31,"p","compact"
"4","audi","a4",2,2008,4,"auto(av)","f",21,30,"p","compact"
"5","audi","a4",2.8,1999,6,"auto(l5)","f",16,26,"p","compact"
"6","audi","a4",2.8,1999,6,"manual(m5)","f",18,26,"p","compact"
"7","audi","a4",3.1,2008,6,"auto(av)","f",18,27,"p","compact"
"8","audi","a4 quattro",1.8,1999,4,"manual(m5)","4",18,26,"p","compact"
"9","audi","a4 quattro",1.8,1999,4,"auto(l5)","4",16,25,"p","compact"
"10","audi","a4 quattro",2,2008,4,"manual(m6)","4",20,28,"p","compact"
"11","audi","a4 quattro",2,2008,4,"auto(s6)","4",19,27,"p","compact"
"12","audi","a4 quattro",2.8,1999,6,"auto(l5)","4",15,25,"p","compact"
"13","audi","a4 quattro",2.8,1999,6,"manual(m5)","4",17,25,"p","compact"
"14","audi","a4 quattro",3.1,2008,6,"auto(s6)","4",17,25,"p","compact"
"15","audi","a4 quattro",3.1,2008,6,"manual(m6)","4",15,25,"p","compact"
"16","audi","a6 quattro",2.8,1999,6,"auto(l5)","4",15,24,"p","midsize"
"17","audi","a6 quattro",3.1,2008,6,"auto(s6)","4",17,25,"p","midsize"
"18","audi","a6 quattro",4.2,2008,8,"auto(s6)","4",16,23,"p","midsize"
"19","chevrolet","c1500 suburban 2wd",5.3,2008,8,"auto(l4)","r",14,20,"r","suv"
"20","chevrolet","c1500 suburban 2wd",5.3,2008,8,"auto(l4)","r",11,15,"e","suv"
"21","chevrolet","c1500 suburban 2wd",5.3,2008,8,"auto(l4)","r",14,20,"r","suv"
"22","chevrolet","c1500 suburban 2wd",5.7,1999,8,"auto(l4)","r",13,17,"r","suv"
"23","chevrolet","c1500 suburban 2wd",6,2008,8,"auto(l4)","r",12,17,"r","suv"
"24","chevrolet","corvette",5.7,1999,8,"manual(m6)","r",16,26,"p","2seater"
```

- fl : fuel (e = ethanol E85, d = diesel, r = regular, p = premium, c = CNG)
- hwy : highway mpg
- manufacturer : automobile manufacturer
- model : model of car
- trans : type of transmission
- year : model year

First, let's import the CSV module, which will assist us in reading in our CSV file. Using some iPython magic, let's set the floating point precision for printing to 2.

Next let's read in our mpg.csv using `csv.DictReader` and convert it to a list of dictionaries. Let's look at the first three elements of our list. We can see that the dictionaries in the list have the column names of the CSV as keys, and data for each specific car are the values.

```
In [53]: import csv

         %precision 2

         with open('mpg.csv') as csvfile:
             mpg = list(csv.DictReader(csvfile))

         mpg[:3] # The first three dictionaries in our list.

Out[53]: [OrderedDict([('', '1'),
                       ('manufacturer', 'audi'),
                       ('model', 'a4'),
                       ('displ', '1.8'),
                       ('year', '1999'),
                       ('cyl', '4'),
                       ('trans', 'auto(l5)'),
                       ('drv', 'f'),
                          ('cty', '18'),
                          ('hwy', '29'),
                          ('fl', 'p'),
                          ('class', 'compact')]),
             OrderedDict([('', '2'),
                          ('manufacturer', 'audi'),
                          ('model', 'a4'),
                          ('displ', '1.8'),
                          ('year', '1999'),
                          ('cyl', '4'),
                          ('trans', 'manual(m5)'),
                          ('drv', 'f'),
                          ('cty', '21'),
                          ('hwy', '29'),
                          ('fl', 'p'),
                          ('class', 'compact')]),
             OrderedDict([('', '3'),
                          ('manufacturer', 'audi'),
                          ('model', 'a4'),
                          ('displ', '2'),
                          ('year', '2008'),
                          ('cyl', '4'),
                          ('trans', 'manual(m6)'),
                          ('drv', 'f'),
                          ('cty', '20'),
                          ('hwy', '31'),
                          ('fl', 'p'),
                          ('class', 'compact')])]
```

`csv.Dictreader` has read in each row of our csv file as a dictionary. `len` shows that our list is comprised of 234 dictionaries.

```
In [40]: len(mpg)
```

```
Out[40]: 234
```

`keys` gives us the column names of our csv.

```
In [41]: mpg[0].keys()
```

```
Out[41]: odict_keys(['', 'manufacturer', 'model', 'displ', 'year', 'cyl', 'trans', 'drv', 'cty',
```

This is how to **find the average `cty` fuel economy across all cars**. All values in the dictionaries are strings, so we need to convert to float.

```
In [42]: sum(float(d['cty']) for d in mpg) / len(mpg)
```

```
Out[42]: 16.86
```

Similarly this is how to **find the average `hwy` fuel economy across all cars**.

```
In [43]: sum(float(d['hwy']) for d in mpg) / len(mpg)
```

```
Out[43]: 23.44
```

Now, let's look at a more complex example. Say **we want to know what the average city MPG is, grouped by the number of cylinders a car has**.

Use set to return the unique values for the number of cylinders the cars in our dataset have.

```
In [44]: cylinders = set(d['cyl'] for d in mpg)
         cylinders
```

```
Out[44]: {'4', '5', '6', '8'}
```

We'll create an empty list where we'll store our calculations. Next, we'll iterate over all the cylinder levels. And then we'll iterate over all the dictionaries.

```
In [45]: CtyMpgByCyl = []

         for c in cylinders: # iterate over all the cylinder levels
             summpg = 0
             cyltypecount = 0
             for d in mpg: # iterate over all dictionaries
                 if d['cyl'] == c: # if the cylinder level type matches,
                     summpg += float(d['cty']) # add the cty mpg
                     cyltypecount += 1 # increment the count
             CtyMpgByCyl.append((c, summpg / cyltypecount)) # append the tuple ('cylinder', 'avg

         CtyMpgByCyl.sort(key=lambda x: x[0])
         CtyMpgByCyl
```

```
Out[45]: [('4', 21.01), ('5', 20.50), ('6', 16.22), ('8', 12.57)]
```

Let's look at one more similar example. Suppose we're interested in **finding the average highway MPG for the different vehicle classes**. Looking at the different vehicle classes, we have 2seater, compact, midsize, minivan, pickup, subcompact, and SUV.

```
In [46]: vehicleclass = set(d['class'] for d in mpg) # what are the class types
         vehicleclass
```

```
Out[46]: {'2seater', 'compact', 'midsize', 'minivan', 'pickup', 'subcompact', 'suv'}
```

Similar to the last example, we iterate over all the vehicle classes, then iterate over all the dictionaries

```
In [47]: HwyMpgByClass = []

         for t in vehicleclass: # iterate over all the vehicle classes
             summpg = 0
             vclasscount = 0
             for d in mpg: # iterate over all dictionaries
```

```
                if d['class'] == t: # if the cylinder amount type matches,
                    summpg += float(d['hwy']) # add the hwy mpg
                    vclasscount += 1 # increment the count
            HwyMpgByClass.append((t, summpg / vclasscount)) # append the tuple ('class', 'avg m

        HwyMpgByClass.sort(key=lambda x: x[1])
        HwyMpgByClass
```

```
Out[47]: [('pickup', 16.88),
          ('suv', 18.13),
          ('minivan', 22.36),
          ('2seater', 24.80),
          ('midsize', 27.29),
          ('subcompact', 28.14),
          ('compact', 28.30)]
```

8 Numerical Python (NumPy)

Numpy is a package widely used in the data science community which lets us work efficiently with arrays and matrices in Python.

First, let's import numpy as np. This lets us use the shortcut np to refer to numpy.

```
In [3]: import numpy as np
```

8.1 Creating Arrays

Create a list and convert it to a numpy array

```
In [4]: mylist = [1, 2, 3]
        x = np.array(mylist)
        x
```

```
Out[4]: array([1, 2, 3])
```

Or just pass in a list directly

```
In [5]: y = np.array([4, 5, 6])
        y
```

```
Out[5]: array([4, 5, 6])
```

Pass in a list of lists to create a multidimensional array.

```
In [6]: m = np.array([[7, 8, 9], [10, 11, 12]])
        m
```

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

Use the shape method to find the dimensions of the array. (rows, columns)

```
In [7]: m.shape

Out[7]: (2, 3)
```

arange returns evenly spaced values within a given interval.

```
In [8]: n = np.arange(0, 30, 2) # start at 0 count up by 2, stop before 30
        n

Out[8]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

reshape returns an array with the same data with a new shape.

```
In [9]: n = n.reshape(3, 5) # reshape array to be 3x5
        n

Out[9]: array([[ 0,  2,  4,  6,  8],
               [10, 12, 14, 16, 18],
               [20, 22, 24, 26, 28]])
```

linspace returns evenly spaced numbers over a specified interval.

```
In [10]: o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
         o

Out[10]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. ])
```

resize changes the shape and size of array in-place.

```
In [11]: o.resize(3, 3)
         o

Out[11]: array([[0. , 0.5, 1. ],
                [1.5, 2. , 2.5],
                [3. , 3.5, 4. ]])
```

ones returns a new array of given shape and type, filled with ones.

```
In [12]: np.ones((3, 2))

Out[12]: array([[1., 1.],
                [1., 1.],
                [1., 1.]])
```

zeros returns a new array of given shape and type, filled with zeros.

```
In [13]: np.zeros((2, 3))

Out[13]: array([[0., 0., 0.],
                [0., 0., 0.]])
```

eye returns a 2-D array with ones on the diagonal and zeros elsewhere.

```
In [14]: np.eye(3)

Out[14]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

diag extracts a diagonal or constructs a diagonal array.

```
In [15]: print(y)
         np.diag(y)

[4 5 6]


Out[15]: array([[4, 0, 0],
                [0, 5, 0],
                [0, 0, 6]])
```

Create an array using repeating list (or see `np.tile`)

```
In [16]: np.array([1, 2, 3] * 3)

Out[16]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Repeat elements of an array using `repeat`.

```
In [17]: np.repeat([1, 2, 3], 3)

Out[17]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

8.2 Combining Arrays

```
In [18]: p = np.ones([2, 3], int)
         p

Out[18]: array([[1, 1, 1],
                [1, 1, 1]])
```

Use `vstack` to stack arrays in sequence vertically (row wise).

```
In [19]: np.vstack([p, 2*p])

Out[19]: array([[1, 1, 1],
                [1, 1, 1],
                [2, 2, 2],
                [2, 2, 2]])
```

Use `hstack` to stack arrays in sequence horizontally (column wise).

```
In [20]: np.hstack([p, 2*p])

Out[20]: array([[1, 1, 1, 2, 2, 2],
                [1, 1, 1, 2, 2, 2]])
```

8.3 Operations

Use +, -, *, / and ** to perform element wise addition, subtraction, multiplication, division and power.

```
In [21]: print(x)
         print(y)
         print(x + y) # elementwise addition     [1 2 3] + [4 5 6] = [5  7  9]
         print(x - y) # elementwise subtraction  [1 2 3] - [4 5 6] = [-3 -3 -3]
```

```
[1 2 3]
[4 5 6]
[5 7 9]
[-3 -3 -3]
```

```
In [22]: print(x * y) # elementwise multiplication  [1 2 3] * [4 5 6] = [4  10  18]
         print(x / y) # elementwise divison         [1 2 3] / [4 5 6] = [0.25  0.4  0.5]
```

```
[ 4 10 18]
[0.25 0.4  0.5 ]
```

```
In [23]: print(x**2) # elementwise power  [1 2 3] ^2 =  [1 4 9]
```

```
[1 4 9]
```

Use >, <, <=, >=, == and != to perform element wise comparison.

```
In [44]: x < y

Out[44]: array([ True,  True,  True])

In [45]: sum(x > 1)

Out[45]: 2
```

Dot Product:
\$
$\begin{bmatrix} x_1 \; x_2 \; x_3 \end{bmatrix}$
.
$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$
= x_1 y_1 + x_2 y_2 + x_3 y_3\$

```
In [24]: x.dot(y) # dot product  1*4 + 2*5 + 3*6

Out[24]: 32
```

For 2-D arrays it is the matrix product:

```
In [41]: a = [[1, 0], [0, 1]]
         b = [[4, 1], [2, 2]]
         np.dot(a, b)
```

```
Out[41]: array([[4, 1],
                [2, 2]])
```

Let's look at transposing arrays. Transposing permutes the dimensions of the array.

```
In [25]: print(y)
         z = np.array([y, y**2])
         z
```

```
[4 5 6]
```

```
Out[25]: array([[ 4,  5,  6],
                [16, 25, 36]])
```

The shape of array z is `(2,3)` before transposing.

```
In [26]: z.shape
```

```
Out[26]: (2, 3)
```

Use `.T` to get the transpose.

```
In [27]: z.T
```

```
Out[27]: array([[ 4, 16],
                [ 5, 25],
                [ 6, 36]])
```

The number of rows has swapped with the number of columns.

```
In [28]: z.T.shape
```

```
Out[28]: (3, 2)
```

Use `.dtype` to see the data type of the elements in the array.

```
In [29]: z.dtype
```

```
Out[29]: dtype('int64')
```

Use `.astype` to cast to a specific type.

```
In [32]: z = z.astype('f')
         print(z.dtype)
         z
```

```
float32
```

```
Out[32]: array([[ 4.,  5.,  6.],
                [16., 25., 36.]], dtype=float32)
```

8.4 Math Functions

Numpy has many built in math functions that can be performed on arrays.

```
In [33]: a = np.array([-4, -2, 1, 3, 5])

In [34]: a.sum()

Out[34]: 3

In [35]: a.max()

Out[35]: 5

In [36]: a.min()

Out[36]: -4

In [37]: a.mean()

Out[37]: 0.6

In [38]: a.std()

Out[38]: 3.2619012860600183

In [43]: np.sin(a)

Out[43]: array([[0.84147098, 0.        ],
                [0.        , 0.84147098]])
```

`argmax` and `argmin` return the index of the maximum and minimum values in the array.

```
In [39]: a.argmax()

Out[39]: 4

In [40]: a.argmin()

Out[40]: 0
```

8.5 Indexing / Slicing

Let's create an array with the squares of 0 through 12

```
In [105]: s = np.arange(13)**2
          s

Out[105]: array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144])
```

Use bracket notation to get the value at a specific index. Remember that indexing starts at 0.

```
In [106]: s[0], s[4], s[-1]
```

```
Out[106]: (0, 16, 144)
```

Use : to indicate a range. `array[start:stop]`
Leaving `start` or `stop` empty will default to the beginning/end of the array.

```
In [107]: s[1:5]
```

```
Out[107]: array([ 1,  4,  9, 16])
```

Use negatives to count from the back.

```
In [108]: s[-4:]
```

```
Out[108]: array([ 81, 100, 121, 144])
```

A second : can be used to indicate step-size. `array[start:stop:stepsize]`
Here we are starting 5th element from the end, and counting backwards by 2 until the beginning of the array is reached.

```
In [109]: s[-5::-2]
```

```
Out[109]: array([64, 36, 16,  4,  0])
```

Let's look at a multidimensional array.

```
In [110]: r = np.arange(36)
          r.resize((6, 6))
          r
```

```
Out[110]: array([[ 0,  1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10, 11],
                 [12, 13, 14, 15, 16, 17],
                 [18, 19, 20, 21, 22, 23],
                 [24, 25, 26, 27, 28, 29],
                 [30, 31, 32, 33, 34, 35]])
```

Use bracket notation to slice: `array[row, column]`

```
In [111]: r[2, 2]
```

```
Out[111]: 14
```

And use : to select a range of rows or columns

```
In [112]: r[3, 3:6]
```

```
Out[112]: array([21, 22, 23])
```

Here we are selecting all the rows up to (and not including) row 2, and all the columns up to (and not including) the last column.

```
In [113]: r[:2, :-1]

Out[113]: array([[ 0,  1,  2,  3,  4],
                 [ 6,  7,  8,  9, 10]])
```

This is a slice of the last row, and only every other element.

```
In [114]: r[-1, ::2]

Out[114]: array([30, 32, 34])
```

We can also perform conditional indexing. Here we are selecting values from the array that are greater than 30. (Also see `np.where`)

```
In [115]: r[r > 30]

Out[115]: array([31, 32, 33, 34, 35])
```

Here we are assigning all values in the array that are greater than 30 to the value of 30.

```
In [116]: r[r > 30] = 30
          r

Out[116]: array([[ 0,  1,  2,  3,  4,  5],
                 [ 6,  7,  8,  9, 10, 11],
                 [12, 13, 14, 15, 16, 17],
                 [18, 19, 20, 21, 22, 23],
                 [24, 25, 26, 27, 28, 29],
                 [30, 30, 30, 30, 30, 30]])
```

8.6 Copying Data

Be careful with copying and modifying arrays in NumPy!

`r2` is a slice of `r`

```
In [117]: r2 = r[:3,:3]
          r2

Out[117]: array([[ 0,  1,  2],
                 [ 6,  7,  8],
                 [12, 13, 14]])
```

Set this slice's values to zero ([:] selects the entire array)

```
In [118]: r2[:] = 0
          r2

Out[118]: array([[0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 0]])
```

r has also been changed!

```
In [119]: r

Out[119]: array([[ 0,  0,  0,  3,  4,  5],
                 [ 0,  0,  0,  9, 10, 11],
                 [ 0,  0,  0, 15, 16, 17],
                 [18, 19, 20, 21, 22, 23],
                 [24, 25, 26, 27, 28, 29],
                 [30, 30, 30, 30, 30, 30]])
```

To avoid this, use `r.copy` to create a copy that will not affect the original array

```
In [120]: r_copy = r.copy()
          r_copy

Out[120]: array([[ 0,  0,  0,  3,  4,  5],
                 [ 0,  0,  0,  9, 10, 11],
                 [ 0,  0,  0, 15, 16, 17],
                 [18, 19, 20, 21, 22, 23],
                 [24, 25, 26, 27, 28, 29],
                 [30, 30, 30, 30, 30, 30]])
```

Now when r_copy is modified, r will not be changed.

```
In [121]: r_copy[:] = 10
          print(r_copy, '\n')
          print(r)

[[10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]
 [10 10 10 10 10 10]]

[[ 0  0  0  3  4  5]
 [ 0  0  0  9 10 11]
 [ 0  0  0 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 30 30 30 30 30]]
```

8.7 Iterating Over Arrays

Let's create a new 4 by 3 array of random numbers 0-9.

```
In [122]: test = np.random.randint(0, 10, (4,3))
          test
```

```
Out[122]: array([[4, 5, 9],
                 [5, 1, 3],
                 [0, 6, 4],
                 [2, 6, 9]])
```

Iterate by row:

```
In [123]: for row in test:
              print(row)
```

```
[4 5 9]
[5 1 3]
[0 6 4]
[2 6 9]
```

We can iterate by row index by using the `len` function on `test`, which returns the number of rows

```
In [124]: for i in range(len(test)):
              print(test[i])
```

```
[4 5 9]
[5 1 3]
[0 6 4]
[2 6 9]
```

We can iterate by using `enumerate`, which gives us the row and the index of the row

```
In [125]: for i, row in enumerate(test):
              print('row', i, 'is', row)
```

```
row 0 is [4 5 9]
row 1 is [5 1 3]
row 2 is [0 6 4]
row 3 is [2 6 9]
```

Use `zip` to iterate over multiple iterables.

```
In [126]: test2 = test**2
          test2
```

```
Out[126]: array([[16, 25, 81],
                 [25,  1,  9],
                 [ 0, 36, 16],
                 [ 4, 36, 81]])
```

```
In [127]: for i, j in zip(test, test2):
              print(i,'+',j,'=',i+j)
```

```
[4 5 9] + [16 25 81] = [20 30 90]
[5 1 3] + [25  1  9] = [30  2 12]
[0 6 4] + [ 0 36 16] = [ 0 42 20]
[2 6 9] + [ 4 36 81] = [ 6 42 90]
```

9 Additional Python Resources

- Coursera: Applied Data Science with Python Specialization
- Python Docs (for general Python documentation)
- Python Classes Docs
- Scipy (for IPython, Numpy, Pandas, and Matplotlib)
- Don't forget to check Stack Overflow!