*You are currently looking at **version 1.1** of this notebook. To download notebooks and datafiles, as well as get help on Jupyter notebooks in the Coursera platform, visit the Jupyter Notebook FAQ (https://www.coursera.org/learn/python-data-analysis/resources/0dhYG) course resource.*

The Python Programming Language: Functions

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

Out[1]: 3

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

Out[2]: 1

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

```
In [10]:  def add_numbers(x,y):
              return x+y

          add_numbers(1, 2)
```

Out[10]: 3

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

```
In [13]:  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
```

add_numbers updated to take an optional flag parameter.

```python
In [25]: 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, 2,3, flag=False))
```

```
Flag is true!
6
```

Assign function add_numbers to variable a.

```python
In [7]: def add_numbers(x,y):
            return x+y

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

Out[7]: 3

The Python Programming Language: Types and Sequences

Use type to return the object's type.

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

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

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

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

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

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

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

Lists are a mutable data structure.

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

Use append to append an object to a list.

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

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

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

Or using the indexing operator:

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

Use + to concatenate lists.

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

Use * to repeat lists.

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

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

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

Now let's look at strings. Use bracket notation to slice a string.

```
In [ ]:  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
```

This will return the last element of the string.

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

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

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

This is a slice from the beginning of the string and stopping before the 3rd element.

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

And this is a slice starting from the 3rd element of the string and going all the way to the end.

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

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

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

split returns a list of all the words in a string, or a list split on a specific character.

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

Make sure you convert objects to strings before concatenating.

In [ ]: `'Chris' + 2`

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

Dictionaries associate keys with values.

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

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

Iterate over all of the keys:

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

Iterate over all of the values:

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

Iterate over all of the items in the list:

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

You can unpack a sequence into different variables:

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

In [ ]: `fname`

In [ ]: `lname`

Make sure the number of values you are unpacking matches the number of variables being assigned.

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

The Python Programming Language: More on Strings

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

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

Python has a built in method for convenient string formatting.

```
In [ ]: 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']))
```

Reading and Writing 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
- 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

```
In [ ]:  import csv

         %precision 2

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

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

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 [ ]:  len(mpg)
```

keys gives us the column names of our csv.

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

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

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

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

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

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

Here's a more complex example where we are grouping the cars by number of cylinder, and finding the average cty mpg for each group.

```
In [ ]:  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'

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

Use set to return the unique values for the class types in our dataset.

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

And here's an example of how to find the average hwy mpg for each class of vehicle in our dataset.

```
In [ ]:  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',

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

The Python Programming Language: Dates and Times

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

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

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

Convert the timestamp to datetime.

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

Handy datetime attributes:

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

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

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

`date.today` returns the current local date.

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

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

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

The Python Programming Language: Objects and map()

An example of a class in python:

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

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

```
In [ ]: person = Person()
        person.set_name('Christopher Brooks')
        person.set_location('Ann Arbor, MI, USA')
        print('{} live in {} and works in the department {}'.format(person.name, person.l
```

Here's an example of mapping the `min` function between two lists.

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

Now let's iterate through the map object to see the values.

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

The Python Programming Language: Lambda and List Comprehensions

Here's an example of lambda that takes in three parameters and adds the first two.

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

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

Let's iterate from 0 to 999 and return the even numbers.

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

Now the same thing but with list comprehension.

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

The Python Programming Language: Numerical Python (NumPy)

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

Creating Arrays

Create a list and convert it to a numpy array

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

Or just pass in a list directly

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

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

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

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

```
In [ ]:  m.shape
```

arange returns evenly spaced values within a given interval.

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

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

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

linspace returns evenly spaced numbers over a specified interval.

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

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

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

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

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

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

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

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

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

diag extracts a diagonal or constructs a diagonal array.

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

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

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

Repeat elements of an array using repeat.

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

Combining Arrays

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

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

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

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

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

Operations

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

```
In [ ]: 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]
```

```
In [ ]: 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]
```

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

Dot Product:

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = x_1 y_1 + x_2 y_2 + x_3 y_3$$

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

```
In [ ]: z = np.array([y, y**2])
        print(len(z)) # number of rows of array
```

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

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

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

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

Use .T to get the transpose.

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

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

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

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

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

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

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

Math Functions

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

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

```
In [ ]:  a.sum()
```

```
In [ ]:  a.max()
```

```
In [ ]:  a.min()
```

```
In [ ]:  a.mean()
```

```
In [ ]:  a.std()
```

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

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

```
In [ ]:  a.argmin()
```

Indexing / Slicing

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

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

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

Use : to indicate a range. `array[start:stop]`

Leaving `start` or `stop` empty will default to the beginning/end of the array.

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

Use negatives to count from the back.

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

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

Let's look at a multidimensional array.

```
In [ ]: r = np.arange(36)
        r.resize((6, 6))
        r
```
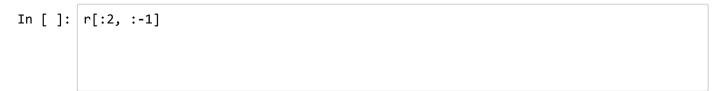
Use bracket notation to slice: `array[row, column]`

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

And use : to select a range of rows or columns

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

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

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

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

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

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

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

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

Copying Data

Be careful with copying and modifying arrays in NumPy!

`r2` is a slice of `r`

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

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

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

`r` has also been changed!

```
In [ ]: r
```

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

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

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

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

Iterating Over Arrays

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

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

Iterate by row:

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

Iterate by index:

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

Iterate by row and index:

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

Use `zip` to iterate over multiple iterables.

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