*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 [20]:

```
x = 1
y = 2
x + y
```

Out[20]:

3

In [21]:

```
x
```

Out[21]:

1

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

In [22]:

```
def add_numbers(x, y):
    return x + y

add_numbers(1, 2)
```

Out[22]:

3

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

In [23]:

```
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.

In [26]:

```
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 = True))
```

```
Flag is true!
6
```

Assign function add_numbers to variable a.

In [27]:

```
def add_numbers(x,y):
    return x+y

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

Out[27]:

```
3
```

The Python Programming Language: Types and Sequences

Use type to return the object's type.

In [28]:

```
type('This is a string')
```

Out[28]:

```
str
```

In [1]:

```
type(None)
```

Out[1]:

```
NoneType
```

In [30]:

```
type(1)
```

Out[30]:

```
int
```

In [31]:

```
type(1.0)
```

Out[31]:

```
float
```

In [32]:

```
type(add_numbers)
```

Out[32]:

```
function
```

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

In [7]:

```
x = (1, 'a', 2, 'b')
type(x)
```

Out[7]:

```
tuple
```

Lists are a mutable data structure.

In [47]:

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

Out[47]:

```
list
```

Use append to append an object to a list.

In [3]:

```
x.append('Hello!')
print(x)
```

```
[1, 'a', 2, 'b', 'Hello!']
```

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

In [5]:

```
for item in x:
    print(item)
```

```
1
a
2
b
Hello!
```

Or using the indexing operator:

In [43]:

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

```
1
a
2
b
3.3
Hello!
```

Use + to concatenate lists.

In [44]:

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

Out[44]:

```
[1, 2, 3, 4]
```

Use * to repeat lists.

In [46]:

```
3*[1]
```

Out[46]:

```
[1, 1, 1]
```

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

In [10]:

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

Out[10]:

```
True
```

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

In [2]:

```
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
```

This will return the last element of the string.

In [6]:

```
x[-1]
```

Out[6]:

```
'g'
```

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

In [21]:

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

Out[21]:

```
'ri'
```

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

In [22]:

```
x[:3]
```

Out[22]:

```
'Thi'
```

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

In [23]:

```
x[3:]
```

Out[23]:

```
's is a string'
```

In [24]:

```
firstname = 'Christopher'
lastname = 'Brooks'

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

```
Christopher Brooks
ChristopherChristopherChristopher
True
```

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

In [41]:

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

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

In [44]:

```
'Chris' + ' ' + str(2)
```

Out[44]:

```
'Chris 2'
```

In [43]:

```
'Chris' + str(2)
```

Out[43]:

```
'Chris2'
```

Dictionaries associate keys with values.

In [7]:

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

Out[7]:

```
'brooksch@umich.edu'
```

In [8]:

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

Iterate over all of the keys:

In [94]:

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

```
None
billg@microsoft.com
brooksch@umich.edu
```

Iterate over all of the values:

In [76]:

```
for email in x.values():
    print(email)
```

```
None
billg@microsoft.com
brooksch@umich.edu
```

Iterate over all of the items in the list:

In [11]:

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

```
Bill Gates
billg@microsoft.com
Christopher Brooks
brooksch@umich.edu
Kevyn Collins-Thompson
None
```

You can unpack a sequence into different variables:

In [9]:

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

In [10]:

```
fname
```

Out[10]:

```
'Christopher'
```

In [16]:

```
email
```

Out[16]:

```
'brooksch@umich.edu'
```

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

In [21]:

```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email, ftest = x
```

In [12]:

```
print('Chris' + 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-82ccfdd3d5d3> in <module>()
----> 1 print('Chris' + 2)

TypeError: Can't convert 'int' object to str implicitly
```

In [2]:

```
print('Chris' + str(2))
```

```
Chris2
```

Python has a built in method for convenient string formatting.

In [22]:

```
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
```

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 [3]:

```
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[3]:

```
[{'': '1',
  'class': 'compact',
  'cty': '18',
  'cyl': '4',
  'displ': '1.8',
  'drv': 'f',
  'fl': 'p',
  'hwy': '29',
  'manufacturer': 'audi',
  'model': 'a4',
  'trans': 'auto(l5)',
  'year': '1999'},
 {'': '2',
  'class': 'compact',
  'cty': '21',
  'cyl': '4',
  'displ': '1.8',
  'drv': 'f'.
```

`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 [24]:

```
len(mpg)
```

Out[24]:

```
234
```

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

In [25]:

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

Out[25]:

```
dict_keys(['', 'manufacturer', 'drv', 'hwy', 'fl', 'trans', 'cyl', 'class',
 'model', 'displ', 'year', '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 [26]:

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

Out[26]:

```
16.86
```

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

In [27]:

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

Out[27]:

```
23.44
```

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

In [4]:

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

Out[4]:

```
{'4', '5', '6', '8'}
```

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 [5]:

```
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 mpg

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

Out[5]:

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

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

In [6]:

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

Out[6]:

```
{'2seater', 'compact', 'midsize', 'minivan', 'pickup', 'subcompact', 'suv'}
```

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

In [7]:

```
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 mpg')

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

Out[7]:

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

The Python Programming Language: Dates and Times

In [12]:

```
import datetime as dt
import time as tm
```

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

In [13]:

```
tm.time()
```

Out[13]:

```
1488561589.60
```

Convert the timestamp to datetime.

In [28]:

```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

Out[28]:

```
datetime.datetime(2017, 3, 3, 17, 26, 16, 324324)
```

Handy datetime attributes:

In [29]:

```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year, mont
```

Out[29]:

```
(2017, 3, 3, 17, 26, 16)
```

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

In [38]:

```
delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

Out[38]:

```
datetime.timedelta(100)
```

`date.today` returns the current local date.

In [39]:

```
today = dt.date.today()
```

In [40]:

```
today - delta # the date 100 days ago
```

Out[40]:

```
datetime.date(2016, 11, 23)
```

In [41]:

```
today > today-delta # compare dates
```

Out[41]:

```
True
```

The Python Programming Language: Objects and map()

An example of a class in python:

In [43]:

```
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 [45]:

```
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.location,
```

```
Christopher Brooks live in Ann Arbor, MI, USA and works in the department Sc
hool of Information.
```

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

In [46]:

```
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[46]:

```
<map at 0x7f8374163ba8>
```

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

In [47]:

```
for item in cheapest:
    print(item)
```

```
9.0
11.0
12.34
2.01
```

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 [48]:

```
my_function = lambda a, b, c : a + b
```

In [49]:

```
my_function(1, 2, 3)
```

Out[49]:

```
3
```

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

In [50]:

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

Out[50]:

```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34.
```

Now the same thing but with list comprehension.

In [51]:

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

Out[51]:

```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34.
```

The Python Programming Language: Numerical Python (NumPy)

In [52]:

```
import numpy as np
```

Creating Arrays

Create a list and convert it to a numpy array

In [53]:

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

Out[53]:

```
array([1, 2, 3])
```

Or just pass in a list directly

In [57]:

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

Out[57]:

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

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

In [58]:

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

Out[58]:

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

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

In [62]:

```
m.shape
```

Out[62]:

```
(2, 3)
```

`arange` returns evenly spaced values within a given interval.

In [63]:

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

Out[63]:

```
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 [64]:

```
n = n.reshape(3, 5) # reshape array to be 3x5
n
```

Out[64]:

```
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 [69]:

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

Out[69]:

```
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 [70]:

```
o.resize(3, 3)
o
```

Out[70]:

```
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 [71]:

```
np.ones((3, 2))
```

Out[71]:

```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

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

In [74]:

```
np.zeros((2, 3))
```

Out[74]:

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

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

In [75]:

```
np.eye(3)
```

Out[75]:

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

`diag` extracts a diagonal or constructs a diagonal array.

In [76]:

```
np.diag(y)
```

Out[76]:

```
array([[4, 0, 0],
       [0, 5, 0],
       [0, 0, 6]])
```

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

In [99]:

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

Out[99]:

```
array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Repeat elements of an array using `repeat`.

In [88]:

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

Out[88]:

```
array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

Combining Arrays

In [91]:

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

Out[91]:

```
array([[1, 1, 1],
       [1, 1, 1]])
```

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

In [92]:

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

Out[92]:

```
array([[1, 1, 1],
       [1, 1, 1],
       [2, 2, 2],
       [2, 2, 2]])
```

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

In [93]:

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

Out[93]:

```
array([[1, 1, 1, 2, 2, 2],
       [1, 1, 1, 2, 2, 2]])
```

Operations

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

In [100]:

```
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]
```

```
[5 7 9]
[-3 -3 -3]
```

In [101]:

```
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 [102]:

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

```
[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 [103]:

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

Out[103]:

```
32
```

In [106]:

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

```
2
```

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

In [120]:

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

Out[120]:

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

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

In [108]:

```
z.shape
```

Out[108]:

```
(2, 3)
```

Use `.T` to get the transpose.

In [109]:

```
z.T
```

Out[109]:

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

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

In [114]:

```
z.T.shape
```

Out[114]:

```
(3, 2)
```

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

In [121]:

```
z.dtype
```

Out[121]:

```
dtype('int64')
```

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

In [118]:

```
z = z.astype('f')
z.dtype
```

Out[118]:

```
dtype('float32')
```

Math Functions

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

In [122]:

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

In [123]:

```
a.sum()
```

Out[123]:

```
3
```

In [124]:

```
a.max()
```

Out[124]:

```
5
```

In [125]:

```
a.min()
```

Out[125]:

```
-4
```

In [126]:

```
a.mean()
```

Out[126]:

```
0.60
```

In [127]:

```
a.std()
```

Out[127]:

```
3.26
```