Week 1

May 30, 2020

---

*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 course resource.*

---

1 The Python Programming Language: Functions

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

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

        add_numbers(1, 2)
```

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

```
In [8]: def add_numbers(x,y,*z):
                return x+y+z

        print(add_numbers(1, 2, 3, 3))
        print(add_numbers(1, 2, 3))


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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-8-9143325c0f90> in <module>()
          2         return x+y+z
          3
    ----> 4 print(add_numbers(1, 2, 3, 3))
          5 print(add_numbers(1, 2, 3))


        <ipython-input-8-9143325c0f90> in add_numbers(x, y, *z)
```

```
      1 def add_numbers(x,y,*z):
----> 2         return x+y+z
      3
      4 print(add_numbers(1, 2, 3, 3))
      5 print(add_numbers(1, 2, 3))


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

add_numbers updated to take an optional flag parameter.

```
In [9]: 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
```

Assign function add_numbers to variable a.

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

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

The Python Programming Language: Types and Sequences
Use type to return the object's type.

```
In [11]: type('This is a string')
```

```
Out[11]: str
```

```
In [10]: type(None)
```

```
Out[10]: NoneType
```

```
In [12]: type(1)
```

```
Out[12]: int
```

```
In [13]: type(1.0)
```

```
Out[13]: float

In [14]: type(add_numbers)

Out[14]: function
```

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

```
In [20]: x = (1, 'a', 2, 'b')
         type(x)

         type(x[0])

Out[20]: int
```

Lists are a mutable data structure.

```
In [19]: x = [1, 'a', 2, 'b']
         type(x)

Out[19]: list
```

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 4th 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 first ele
        lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the last ele
        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.com'}
        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 [23]: print('Chris', str(2), sep = " ")

Chris 2
```

Python has a built in method for convenient string formatting.

```
In [26]: sales_record = {
         'price': 3.24,
         'num_items': 4,
         'person': 'Chris'}

         sales_statement = '{'person'} bought {'num_items'} item(s) at a price of {'price'} each

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

```
  File "<ipython-input-26-138f51d3189e>", line 6
    sales_statement = '{'person'} bought {'num_items'} item(s) at a price of {'price'} each
                                  ^
SyntaxError: invalid syntax
```

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 [27]:
```
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[27]:
```
[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 [28]: len(mpg)
```

```
Out[28]: 234
```

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

```
In [29]: mpg[0].keys()
```

```
Out[29]: 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 [30]: sum(float(d['cty']) for d in mpg) / len(mpg)
```

```
Out[30]: 16.86
```

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

         cylinders.add('8')
         print(cylinders)
```

```
{'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 [35]: 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[35]: [('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 [ ]: 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', 'avg mp

        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 year, m
```

`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.location
```

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

```
In [37]: 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[37]: <map at 0x7f48b0070cc0>
```

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

```
In [38]: 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 [ ]: 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 [40]: import numpy as np
```

Creating Arrays
Create a list and convert it to a numpy array

```
In [41]: mylist = [1, 2, 3]
         x = np.array(mylist)
         x

Out[41]: array([1, 2, 3])
```

Or just pass in a list directly

```
In [43]: y = np.array([4, 5, 6], dtype = 'int32')
         y

Out[43]: array([4, 5, 6], dtype=int32)
```

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

```
In [44]: m = np.array([[7, 8, 9], [10, 11, 12]])
         m

Out[44]: array([[ 7,  8,  9],
                [10, 11, 12]])
```

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

```
In [45]: m.shape

Out[45]: (2, 3)
```

arange returns evenly spaced values within a given interval.

```
In [47]: n = np.arange(0, 30, 2, dtype = 'int64') # start at 0 count up by 2, stop before 30
         n

Out[47]: 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 [ ]: n = n.reshape(3, 5) # reshape array to be 3x5
        n
```

linspace returns evenly spaced numbers over a specified interval.

```
In [48]: o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
         o

Out[48]: 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 [57]: c = o
         c.resize(3, 3)
         c

Out[57]: array([[ 0. ,  0.5,  1. ],
                [ 1.5,  2. ,  2.5],
                [ 3. ,  3.5,  4. ]])

In [61]: o.reshape(3,3)
         o.astype('int')

Out[61]: array([[0, 0, 1],
                [1, 2, 2],
                [3, 3, 4]])
```

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

```
In [54]: np.ones((3, 2))
```

```
Out[54]: array([[ 1.,  1.],
                [ 1.,  1.],
                [ 1.,  1.]])
```

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

```
In [55]: np.zeros((2, 3))

Out[55]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])
```

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

```
In [62]: np.eye(3)

Out[62]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

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

```
In [69]: np.diag(y)

Out[69]: array([[4, 0, 0],
                [0, 5, 0],
                [0, 0, 6]], dtype=int32)
```

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

```
In [63]: np.array([1, 2, 3] * 3)

Out[63]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])

In [68]: [1, 2, 3] * 2

Out[68]: [1, 2, 3, 1, 2, 3]
```

Repeat elements of an array using `repeat`.

```
In [70]: np.repeat([1, 2, 3], 3)

Out[70]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

Combining Arrays

```
In [86]: p = np.ones([2, 3], int)
         p

Out[86]: array([[1, 1, 1],
                [1, 1, 1]])
```

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