*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.*

The Python Programming Language: Functions

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

Out[2]: 3

In [3]:
```
y
```

Out[3]: 2

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

In [4]:
```
def add_numbers(x, y):
    return x + y

add_numbers(1, 2)
```

Out[4]: 3

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

In [5]:
```
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 [6]:
```
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 [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.

In [8]:
```
type('This is a string')
```

Out[8]: str

In [9]:
```
type(None)
```

Out[9]: NoneType

In [10]:
```
type(1)
```

Out[10]: int

In [11]:
```
type(1.0)
```

Out[11]:
```
float
```

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

Out[12]:
```
function
```

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

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

Out[13]:
```
tuple
```

Lists are a mutable data structure.

In [14]:
```
x = [1, 'a', 2, 'b']
type(x)
```

Out[14]:
```
list
```

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

In [15]:
```
x.append(3.3)
print(x)
```

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

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

In [16]:
```
for item in x:
    print(item)
```

```
1
a
2
b
3.3
```

Or using the indexing operator:

In [17]:
```
i=0
while( i != len(x) ):
    print(x[i])
    i = i + 1
```

```
1
a
2
b
3.3
```

Use `+` to concatenate lists.

In [18]:
```
[1,2] + [3,4]
```

Out[18]:
```
[1, 2, 3, 4]
```

Use `*` to repeat lists.

In [19]:
```
[1]*3
```

Out[19]:
```
[1, 1, 1]
```

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

In [20]:
```
1 in [1, 2, 3]
```

Out[20]:
```
True
```

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

In [21]:
```
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 [22]:
```
x[-1]
```

Out[22]:
```
'g'
```

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

In [23]:
```
x[-4:-2]
```

Out[23]:
```
'ri'
```

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

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

Out[24]:
```
'Thi'
```

And this is a slice starting from the 4th element of the string and going all the way to the end.

In [25]:
```
x[3:]
```

Out[25]:
```
's is a string'
```

In [26]:
```
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 [27]:
```
firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the first element of the list
lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the last element of the list
print(firstname)
print(lastname)
```

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

In [28]:
```
'Chris' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-9d01956b24db> in <module>
----> 1 'Chris' + 2

TypeError: can only concatenate str (not "int") to str
```

In [34]:
```
'Chris' + str(2)
```

Out[34]:
```
'Chris2'
```

Dictionaries associate keys with values.

In [37]:
```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.com'}
x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

Out[37]:
```
'brooksch@umich.edu'
```

In [38]:
```
x['Kevyn Collins-Thompson'] = None
x['Kevyn Collins-Thompson']
```

Iterate over all of the keys:

In [39]:
```
for name in x:
    print(x[name])
```

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

Iterate over all of the values:

In [40]:
```
for email in x.values():
    print(email)
```

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

Iterate over all of the items in the list:

In [41]:
```
for name, email in x.items():
    print(name)
    print(email)
```

```
Christopher Brooks
brooksch@umich.edu
Bill Gates
```

```
billg@microsoft.com
Kevyn Collins-Thompson
None
```

You can unpack a sequence into different variables:

In [42]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

In [43]:
```
fname
```

Out[43]:
```
'Christopher'
```

In [44]:
```
lname
```

Out[44]:
```
'Brooks'
```

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

In [45]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email = x
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-45-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)
```

The Python Programming Language: More on Strings

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

In [46]:
```
print('Chris' + str(2))
```

```
Chris2
```

Python has a built in method for convenient string formatting.

In [47]:
```
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 [53]:
```
import csv

%precision 2

with open('../datasets/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)'),
```

`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 [54]:
```
len(mpg)
```

Out[54]:
```
234
```

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

In [55]:
```
mpg[0].keys()
```

Out[55]:
```
odict_keys(['', 'manufacturer', 'model', 'displ', 'year', 'cyl', 'trans', 'drv', 'cty', 'hwy', 'fl', 'class'])
```

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 [56]:
```
sum(float(d['cty']) for d in mpg) / len(mpg)
```

Out[56]:
```
16.86
```

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

In [57]:
```
sum(float(d['hwy']) for d in mpg) / len(mpg)
```

Out[57]:
```
23.44
```

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

In [58]:
```
cylinders = set(d['cyl'] for d in mpg)
cylinders
```

Out[58]:
```
{'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 [59]:
```
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[59]:
```
[('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 [60]:
```
vehicleclass = set(d['class'] for d in mpg) # what are the class types
vehicleclass
```

Out[60]:
```
{'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 [61]:
```
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[61]:
```
[('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 [62]:
```
import datetime as dt
import time as tm
```

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

In [63]:
```
tm.time()
```

Out[63]:
```
1603101455.45
```

Convert the timestamp to datetime.

In [64]:
```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

Out[64]:
```
datetime.datetime(2020, 10, 19, 9, 57, 35, 466040)
```

Handy datetime attributes:

In [65]:
```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year, month, day, etc.from a datetime
```

Out[65]:
```
(2020, 10, 19, 9, 57, 35)
```

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

In [66]:
```
delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

Out[66]:
```
datetime.timedelta(days=100)
```

`date.today` returns the current local date.

In [67]:
```
today = dt.date.today()
```

In [68]:
```
today - delta # the date 100 days ago
```

Out[68]:
```
datetime.date(2020, 7, 11)
```

In [69]:
```
today > today-delta # compare dates
```

Out[69]:
```
True
```

The Python Programming Language: Objects and map()

An example of a class in python:

In [70]:
```
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 [71]:
```
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, person.department))
```

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

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

In [72]:
```
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[72]:
```
<map at 0x7f69185fe438>
```

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

In [73]:
```
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 [74]:
```
my_function = lambda a, b, c : a + b
```

In [75]:
```
my_function(1, 2, 3)
```

Out[75]:
```
3
```

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

In [76]:
```
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list
```

Out[76]:
```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
```

Now the same thing but with list comprehension.

In [77]:
```
my_list = [number for number in range(0,1000) if number % 2 == 0]
my_list
```

Out[77]:
```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
```