*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)](https://www.coursera.org/learn/python-data-analysis/resources/0dhYG) course resource.*

Functions

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

In [1]:
```
def add_numbers(x, y):
    return x + y

add_numbers(1, 2)
```

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

In [2]:
```
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
```

The `add_numbers` updated to take an optional flag parameter.

In [3]:
```
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 [4]:
```
def add_numbers(x,y):
    return x+y

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

Out[4]:
```
3
```

Types and Sequences

Type

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

In [5]:
```
type('This is a string')
```

Out[5]:
```
str
```

In [6]:
```
type(None)
```

Out[6]:
```
NoneType
```

In [7]:
```
type(1)
```

Out[7]:
```
int
```

In [8]:
```
type(1.0)
```

Out[8]:
```
float
```

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

Out[9]:
```
function
```

Tuple

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

In [10]:
```
x = (1, 'a', 2, 'b')
type(x)
```

Out[10]:
```
tuple
```

List

Lists are a mutable data structure.

In [11]:
```
x = [1, 'a', 2, 'b']
type(x)
```

Out[11]:
```
list
```

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

In [12]:
```
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 [13]:
```
for item in x:
    print(item)
```

```
1
a
2
b
3.3
```

Or using the indexing operator:

In [14]:
```
i=0
while( i != len(x) ):
    print(x[i])
    i = i + 1
```

```
1
a
2
b
3.3
```

Use `+` to concatenate lists.

In [15]:
```
[1,2] + [3,4]
```

Out[15]:
```
[1, 2, 3, 4]
```

Use `*` to repeat lists.

In [16]:
```
[1]*3
```

Out[16]:
```
[1, 1, 1]
```

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

In [17]:
```
1 in [1, 2, 3]
```

Out[17]:
```
True
```

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

In [18]:
```
x = 'This is a string'
print(x[0]) #first character
print(x[0:1]) #first character, but we have explicitly set the end characte
r
print(x[0:2]) #first two characters
```

```
T
T
Th
```

This will return the last element of the string.

In [19]:
```
x[-1]
```

Out[19]:
```
'g'
```

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

```
In [20]: x[-4:-2]
```

```
Out[20]: 'ri'
```

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

```
In [21]: x[:3]
```

```
Out[21]: 'Thi'
```

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

```
In [22]: x[3:]
```

```
Out[22]: 's is a string'
```

```
In [23]: firstname = 'Christopher'
         lastname = 'Brooks'

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

```
Christopher Brooks
ChristopherChristopherChristopher
True
```

Function `split` returns a list of all the words in a string, or a list split on a specific character.
split('xx') 按照 'xx' 来进行 string 的分割。

```
In [24]: 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 [25]:
```
'Chris' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-25-9d01956b24db> in <module>
----> 1 'Chris' + 2

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

In [27]:
```
'Chris' + str(2)
```

Out[27]:
```
'Chris2'
```

Dictionary

Dictionaries associate keys with values.

In [28]:
```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@micro
soft.com'}
x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

Out[28]:
```
'brooksch@umich.edu'
```

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

Iterate over all of the keys:

In [30]:
```
for name in x:
    print(name)
    print(x[name])
    print('====')
```

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

Iterate over all of the values:

In [31]:
```
for email in x.values():
    print(email)
```

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

Iterate over all of the items in the list:

In [32]:
```
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 [33]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

In [34]:
```
fname
```

Out[34]:
```
'Christopher'
```

In [35]:
```
lname
```

Out[35]:
```
'Brooks'
```

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

In [36]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email = x
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-36-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)
```

More on Strings

Make sure you convert objects to strings before concatenating.

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-928a1e955b60> in <module>
----> 1 print('Chris' + 2)

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

In [38]:
```
print('Chris' + str(2))
```

```
Chris2
```

Python has a built in method for convenient string formatting.
String 自带的 Function `.format(string[x])` 会自动,依次将 x 的内容填入 string 的 {} 中。

In [39]:
```
sales_record = {
'price': 3.24,
'num_items': 4,
'person': 'Chris'}

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

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

Use csv.DictReader to Read CSV

In [40]:
```
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[40]:
```
[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 [41]:
```
len(mpg)
```

Out[41]:
```
234
```

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

In [42]:
```
mpg[0].keys()
```

Out[42]:
```
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 [43]:
```
sum(float(d['cty']) for d in mpg) / len(mpg)
```

Out[43]:
```
16.86
```

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

In [44]:
```
sum(float(d['hwy']) for d in mpg) / len(mpg)
```

Out[44]:
```
23.44
```

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

In [45]:
```
cylinders = set(d['cyl'] for d in mpg)
cylinders
```

Out[45]:
```
{'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 [46]:
```
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 ('cyl
inder', 'avg mpg')

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

Out[46]:
```
[('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 [47]:
```
vehicleclass = set(d['class'] for d in mpg) # what are the class types
vehicleclass
```

Out[47]:
```
{'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 [48]:
```
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 ('cl
ass', 'avg mpg')

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

Out[48]:
```
[('pickup', 16.88),
 ('suv', 18.13),
 ('minivan', 22.36),
 ('2seater', 24.80),
 ('midsize', 27.29),
 ('subcompact', 28.14),
 ('compact', 28.30)]
```

Use Pandas to Read CSV

In [49]:
```
import pandas as pd

mpg = pd.read_csv('mpg.csv')
mpg.head()
```

Out[49]:

	Unnamed: 0	manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
0	1	audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
1	2	audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
2	3	audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
3	4	audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
4	5	audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact

`len` shows that our list is comprised of 234 dictionaries.

```
In [50]: len(mpg)
```

```
Out[50]: 234
```

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

```
In [51]: mpg.columns
```

```
Out[51]: Index(['Unnamed: 0', 'manufacturer', 'model', 'displ', 'year', 'cyl', 'tran
         s',
                'drv', 'cty', 'hwy', 'fl', 'class'],
               dtype='object')
```

This is how to find the average cty fuel economy across all cars. All columns values in the DataFrame are Series, so we can use it directly.

```
In [52]: print(type(mpg))
         print(type(mpg['cty']))
```

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.series.Series'>
```

```
In [53]: sum(mpg['cty']) / len(mpg)
```

```
Out[53]: 16.86
```

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

```
In [54]: sum(mpg['hwy']) / len(mpg)
```

```
Out[54]: 23.44
```

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

```
In [55]: cylinders = set(mpg['cyl'])
         cylinders
```

```
Out[55]: {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 [56]:
```
CtyMpgByCyl = []

for c in cylinders: # iterate over all the cylinder levels
    summpg = 0
    cyltypecount = 0
    summpg = sum(mpg[mpg['cyl'] == c]['cty']) # if the cylinder level type
 matches, add the cty mpg
    cyltypecount = sum(mpg['cyl'] == c) # sum of all TRUE for each type cyl
inder. In order to count each type cylinder
    CtyMpgByCyl.append((str(c), summpg / cyltypecount))

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

Out[56]:
```
[('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 [57]:
```
vehicleclass = set(mpg['class']) # what are the class types
vehicleclass
```

Out[57]:
```
{'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 [58]:
```
HwyMpgByClass = []

for t in vehicleclass: # iterate over all the vehicle classes
    summpg = 0
    vclasscount = 0
    summpg = sum(mpg[mpg['class'] == t]['hwy']) # if the cylinder amount ty
pe matches, add the hwy mpg
    vclasscount = sum(mpg['class'] == t) # sum of all TRUE for each class.
 In order to count in each class
    HwyMpgByClass.append((t, summpg / vclasscount)) # append the tuple ('cl
ass', 'avg mpg')

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

Out[58]:
```
[('pickup', 16.88),
 ('suv', 18.13),
 ('minivan', 22.36),
 ('2seater', 24.80),
 ('midsize', 27.29),
 ('subcompact', 28.14),
 ('compact', 28.30)]
```

Dates and Times

In [59]:
```
import datetime as dt
import time as tm
```

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

In [60]:
```
tm.time()
```

Out[60]:
```
1566314704.26
```

Convert the timestamp to datetime.

In [61]:
```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

Out[61]:
```
datetime.datetime(2019, 8, 20, 11, 25, 4, 670063)
```

Handy datetime attributes:

In [62]:
```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second
# get year, month, day, etc.from a datetime
```

Out[62]:
```
(2019, 8, 20, 11, 25, 4)
```

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

In [63]:
```
delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

Out[63]:
```
datetime.timedelta(days=100)
```

`date.today` returns the current local date.

In [64]:
```
today = dt.date.today()
today
```

Out[64]:
```
datetime.date(2019, 8, 20)
```

In [65]:
```
today - delta # the date 100 days ago
```

Out[65]:
```
datetime.date(2019, 5, 12)
```

In [66]:
```
today > today-delta # compare dates
```

Out[66]:
```
True
```

Objects and map()

An example of a class in python:

In [67]:
```
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 [68]:
```
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, pe
rson.location, person.department))
```

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

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

In [69]:
```
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[69]:
```
<map at 0x8e837f0>
```

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

In [70]:
```
for item in cheapest:
    print(item)
```

```
9.0
11.0
12.34
2.01
```

和直接用 `min()` 一样

In [71]:
```
min(store1, store2)
```

Out[71]:
```
[9.00, 11.10, 12.34, 2.01]
```

Lambda and List Comprehensions

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

In [72]:
```
my_function = lambda a, b, c : a + b
```

In [73]:
```
my_function(1, 2, 3)
```

Out[73]:
```
3
```

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

In [74]:
```
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 [75]:
```
my_list = [number for number in range(0,1000) if number % 2 == 0]
# my_list
```

Numerical Python (NumPy)

In [76]:
```
import numpy as np
```

Creating Arrays

Create a list and convert it to a numpy array

In [77]:
```
mylist = [1, 2, 3]
x = np.array(mylist)
x
```

Out[77]:
```
array([1, 2, 3])
```

Or just pass in a list directly

In [78]:
```
y = np.array([4, 5, 6])
y
```

Out[78]:
```
array([4, 5, 6])
```

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

In [79]:
```
m = np.array([[7, 8, 9], [10, 11, 12]])
m
```

Out[79]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

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

In [80]:
```
m.shape
```

Out[80]:
```
(2, 3)
```

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

In [81]:
```
n = np.arange(0, 30, 2) # start at 0 count up by 2, stop before 30
n
```

Out[81]:
```
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 [82]:
```
n = n.reshape(3, 5) # reshape array to be 3x5
n
```

Out[82]:
```
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 [83]:
```
o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
o
```

Out[83]:
```
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 [84]:
```
o.resize(3, 3)
o
```

Out[84]:
```
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 [85]:
```
np.ones((3, 2))
```

Out[85]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

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

In [86]:
```
np.zeros((2, 3))
```

Out[86]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

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

In [87]:
```
np.eye(3)
```

Out[87]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

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

In [88]:
```
np.diag(y)
```

Out[88]:
```
array([[4, 0, 0],
       [0, 5, 0],
       [0, 0, 6]])
```

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

In [89]:
```
np.array([1, 2, 3] * 3)
```

Out[89]:
```
array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Repeat elements of an array using `repeat` .

In [90]:
```
np.repeat([1, 2, 3], 3)
```

Out[90]:
```
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 [94]:
```
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 [95]:
```
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 [96]:
```
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 [97]:
```
x.dot(y) # dot product  1*4 + 2*5 + 3*6
```

Out[97]:
```
32
```

In [98]:
```
z = np.array([y, y**2])
print(z)
print(len(z)) # number of rows of array
```

```
[[ 4  5  6]
 [16 25 36]]
2
```

Let's look at transposing arrays. Transposing permutes the dimensions of the array.
The shape of array `z` is `(2,3)` before transposing.

In [99]:
```
z.shape
```

Out[99]:
```
(2, 3)
```

Use `.T` to get the transpose.

In [100]:
```
z.T
```

Out[100]:
```
array([[ 4, 16],
       [ 5, 25],
       [ 6, 36]])
```

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

In [101]:
```
z.T.shape
```

Out[101]:
```
(3, 2)
```

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

In [102]:
```
z.dtype
```

Out[102]:
```
dtype('int32')
```

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

In [103]:
```
z = z.astype('f')
z.dtype
```

Out[103]:
```
dtype('float32')
```

Math Functions

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

In [104]:
```
a = np.array([-4, -2, 1, 3, 5])
```

In [105]:
```
a.sum()
```

Out[105]:
```
3
```

In [106]:
```
a.max()
```

Out[106]:
```
5
```

In [107]:
```
a.min()
```

Out[107]:
```
-4
```

In [108]:
```
a.mean()
```

Out[108]:
```
0.6
```

In [109]:
```
a.std()
```

Out[109]:
```
3.2619012860600183
```

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

In [110]:
```
a.argmax()
```

Out[110]:
```
4
```

In [111]:
```
a.argmin()
```

Out[111]:
```
0
```

Indexing / Slicing

In [112]:
```
s = np.arange(13)**2
s
```

Out[112]:
```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121, 144],
      dtype=int32)
```

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

In [113]:
```
s[0], s[4], s[-1]
```

Out[113]:
```
(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 [114]:
```
s[1:5]
```

Out[114]:
```
array([ 1,  4,  9, 16], dtype=int32)
```

Use negatives to count from the back.

In [115]:
```
s[-4:]
```

Out[115]:
```
array([ 81, 100, 121, 144], dtype=int32)
```

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 [116]:
```
s[-5::-2]
```

Out[116]:
```
array([64, 36, 16,  4,  0], dtype=int32)
```

Let's look at a multidimensional array.

In [117]:
```
r = np.arange(36)
r.resize((6, 6))
r
```

Out[117]:
```
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 [118]:
```
r[2, 2]
```

Out[118]:
```
14
```

And use : to select a range of rows or columns

In [119]:
```
r[3, 3:6]
```

Out[119]:
```
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 [120]:
```
r[:2, :-1]
```

Out[120]:
```
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 [121]:
```
r[-1, ::2]
```

Out[121]:
```
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 [122]:
```
r[r > 30]
```

Out[122]:
```
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 [123]:
```
r[r > 30] = 30
r
```

Out[123]:
```
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]])
```

Copying Data

Be careful with copying and modifying arrays in NumPy!

`r2` is a slice of `r`

In [124]:
```
r2 = r[:3,:3]
r2
```

Out[124]:
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [12, 13, 14]])
```

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

In [125]:
```
r2[:] = 0
r2
```

Out[125]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

`r` has also been changed!

In [126]:
```
r
```

Out[126]:
```
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 [127]:
```
r_copy = r.copy()
r_copy
```

Out[127]:
```
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 [128]:
```
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]]
```

Iterating Over Arrays

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

In [129]:
```
test = np.random.randint(0, 10, (4,3))
test
```

Out[129]:
```
array([[9, 8, 7],
       [1, 5, 9],
       [2, 0, 5],
       [3, 5, 4]])
```

Iterate by row:

In [130]:
```
for row in test:
    print(row)
```

```
[9 8 7]
[1 5 9]
[2 0 5]
[3 5 4]
```

Iterate by index:

In [131]:
```
for i in range(len(test)):
    print(test[i])
```

```
[9 8 7]
[1 5 9]
[2 0 5]
[3 5 4]
```

Iterate by row and index:

In [132]:
```
for i, row in enumerate(test):
    print('row', i, 'is', row)
```

```
row 0 is [9 8 7]
row 1 is [1 5 9]
row 2 is [2 0 5]
row 3 is [3 5 4]
```

Use `zip` to iterate over multiple iterables.

In [133]:
```
test2 = test**2
test2
```

Out[133]:
```
array([[81, 64, 49],
       [ 1, 25, 81],
       [ 4,  0, 25],
       [ 9, 25, 16]], dtype=int32)
```

In [134]:
```
for i, j in zip(test, test2):
    print(i,'+',j,'=',i+j)
    print('--')
```

```
[9 8 7] + [81 64 49] = [90 72 56]
--
[1 5 9] + [ 1 25 81] = [ 2 30 90]
--
[2 0 5] + [ 4  0 25] = [ 6  0 30]
--
[3 5 4] + [ 9 25 16] = [12 30 20]
--
```