Khalil MASMOUDI

Journée Statistique, Mathématique et Data Science 2019

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 [4]:

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

add_numbers(111, 232)
```

Out[4]:

```
343
```

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

In [6]:

```
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))# ceci est un commentaire
```

```
3
6
```

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

In [5]:

```
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 [6]:

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

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

Out[6]:

```
3
```

The Python Programming Language: Types and Sequences

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

In [7]:

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

Out[7]:

```
str
```

In [8]:

```
type(None)
```

Out[8]:

```
NoneType
```

In [9]:

```
type(1)
```

Out[9]:

```
int
```

In [10]:

```
type(1.0)
```

Out[10]:

```
float
```

In [11]:

```
type(add_numbers)
```

Out[11]:

```
function
```

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

In [12]:

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

Out[12]:

```
tuple
```

Lists are a mutable data structure.

In [7]:

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

Out[7]:

```
list
```

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

In [8]:

```
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 [10]:

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

print(len(x))
```

```
1
a
2
b
3.3
5
```

Or using the indexing operator:

In [14]:

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

print('on est en dehors de la boucle while')
```

```
1
```

```
a
2
b
3.3
on est en dehors de la boucle while
```

Use `+` to concatenate lists.

In [17]:

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

Out[17]:

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

Use `*` to repeat lists.

In [15]:

```
[1, 2]*3
```

Out[15]:

```
[1, 2, 1, 2, 1, 2]
```

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

In [23]:

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

Out[23]:

```
False
```

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

In [24]:

```
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 [25]:

```
x[-1]
```

Out[25]:

```
'g'
```

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

In [26]:

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

Out[26]:

```
'ri'
```

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

In [27]:

```
x[:3]
```

Out[27]:

```
'Thi'
```

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

In [28]:

```
x[3:]
```

Out[28]:

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

In [29]:

```
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 [30]:

```
firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the first ele
ment of the list
lastname = 'Christopher,Arthur,Hansen,Brooks'.split(',')[-1] # [-1] selects the last elem
ent of the list
print(firstname)
print(lastname)
```

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

In [27]:

```
'Chris' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-9d01956b24db> in <module>()
----> 1 'Chris' + 2
```

```
TypeError: must be str, not int
```

In [28]:

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

Out[28]:

```
'Chris2'
```

Dictionaries associate keys with values.

In [44]:

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

Out[44]:

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

In [45]:

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

In [42]:

```
x= {"Age": 18, "taille": 1.7}
x["taille"]
```

Out[42]:

```
1.7
```

Iterate over all of the keys:

In [43]:

```
for name in x:
    print(name +": " + str(x[name]))
```

```
Age: 18
taille: 1.7
```

Iterate over all of the values:

In [46]:

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

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

Iterate over all of the items in the list:

In [47]:

```
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 [48]:

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

In [49]:

```
fname
```

Out[49]:

```
'Christopher'
```

In [50]:

```
lname
```

Out[50]:

```
'Brooks'
```

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

In [51]:

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-51-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 [52]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-52-928a1e955b60> in <module>()
----> 1 print('Chris' + 2)

TypeError: must be str, not int
```

In [53]:

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

```
Chris2
```

Python has a built in method for convenient string formatting.

In [40]:

```
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 [54]:

```
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[54]:

```
[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 [55]:

```
len(mpg)
```

Out[55]:

```
234
```

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

In [56]:

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

Out[56]:

```
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 [57]:

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

Out[57]:

```
16.86
```

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

In [58]:

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

Out[58]:

```
23.44
```

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

In [59]:

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

Out[59]:

```
{'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 [47]:

```
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[47]:

```
[('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 [48]:

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

Out[48]:

```
{'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 [49]:

```
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
g')

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

Out[49]:

Out[49]:

```
[('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 [50]:

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

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

In [51]:

```
tm.time()
```

Out[51]:

```
1572685325.20
```

Convert the timestamp to datetime.

In [52]:

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

Out[52]:

```
datetime.datetime(2019, 11, 2, 10, 2, 7, 491539)
```

Handy datetime attributes:

In [53]:

```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year, m
onth, day, etc.from a datetime
```

Out[53]:

```
(2019, 11, 2, 10, 2, 7)
```

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

In [54]:

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

Out[54]:

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

`date.today` **returns the current local date.**

`date.today` returns the current local date.

In [55]:

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

In [56]:

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

Out[56]:

```
datetime.date(2019, 7, 25)
```

In [57]:

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

Out[57]:

```
True
```

The Python Programming Language: Objects and map()

An example of a class in python:

In [50]:

```
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 [51]:

```
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 Infor
mation
```

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

In [52]:

```
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[52]:

```
<map at 0x12998ab05f8>
```

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

In [53]:

```
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 [54]:

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

In [55]:

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

Out[55]:

```
3
```

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

In [56]:

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

Out[56]:

```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
 38,
 40,
 42,
 44,
 46,
 48,
 50,
 52
```

52,
54,
56,
58,
60,
62,
64,
66,
68,
70,
72,
74,
76,
78,
80,
82,
84,
86,
88,
90,
92,
94,
96,
98,
100,
102,
104,
106,
108,
110,
112,
114,
116,
118,
120,
122,
124,
126,
128,
130,
132,
134,
136,
138,
140,
142,
144,
146,
148,
150,
152,
154,
156,
158,
160,
162,
164,
166,
168,
170,
172,
174,
176,
178,
180,
182,
184,
186,
188,
190,
192,
194,
196

196,
198,
200,
202,
204,
206,
208,
210,
212,
214,
216,
218,
220,
222,
224,
226,
228,
230,
232,
234,
236,
238,
240,
242,
244,
246,
248,
250,
252,
254,
256,
258,
260,
262,
264,
266,
268,
270,
272,
274,
276,
278,
280,
282,
284,
286,
288,
290,
292,
294,
296,
298,
300,
302,
304,
306,
308,
310,
312,
314,
316,
318,
320,
322,
324,
326,
328,
330,
332,
334,
336,
338,
340

340,
342,
344,
346,
348,
350,
352,
354,
356,
358,
360,
362,
364,
366,
368,
370,
372,
374,
376,
378,
380,
382,
384,
386,
388,
390,
392,
394,
396,
398,
400,
402,
404,
406,
408,
410,
412,
414,
416,
418,
420,
422,
424,
426,
428,
430,
432,
434,
436,
438,
440,
442,
444,
446,
448,
450,
452,
454,
456,
458,
460,
462,
464,
466,
468,
470,
472,
474,
476,
478,
480,
482,
484

484,
486,
488,
490,
492,
494,
496,
498,
500,
502,
504,
506,
508,
510,
512,
514,
516,
518,
520,
522,
524,
526,
528,
530,
532,
534,
536,
538,
540,
542,
544,
546,
548,
550,
552,
554,
556,
558,
560,
562,
564,
566,
568,
570,
572,
574,
576,
578,
580,
582,
584,
586,
588,
590,
592,
594,
596,
598,
600,
602,
604,
606,
608,
610,
612,
614,
616,
618,
620,
622,
624,
626,
628

628,
630,
632,
634,
636,
638,
640,
642,
644,
646,
648,
650,
652,
654,
656,
658,
660,
662,
664,
666,
668,
670,
672,
674,
676,
678,
680,
682,
684,
686,
688,
690,
692,
694,
696,
698,
700,
702,
704,
706,
708,
710,
712,
714,
716,
718,
720,
722,
724,
726,
728,
730,
732,
734,
736,
738,
740,
742,
744,
746,
748,
750,
752,
754,
756,
758,
760,
762,
764,
766,
768,
770,
772

772,
774,
776,
778,
780,
782,
784,
786,
788,
790,
792,
794,
796,
798,
800,
802,
804,
806,
808,
810,
812,
814,
816,
818,
820,
822,
824,
826,
828,
830,
832,
834,
836,
838,
840,
842,
844,
846,
848,
850,
852,
854,
856,
858,
860,
862,
864,
866,
868,
870,
872,
874,
876,
878,
880,
882,
884,
886,
888,
890,
892,
894,
896,
898,
900,
902,
904,
906,
908,
910,
912,
914,
916

```
 918,
 920,
 922,
 924,
 926,
 928,
 930,
 932,
 934,
 936,
 938,
 940,
 942,
 944,
 946,
 948,
 950,
 952,
 954,
 956,
 958,
 960,
 962,
 964,
 966,
 968,
 970,
 972,
 974,
 976,
 978,
 980,
 982,
 984,
 986,
 988,
 990,
 992,
 994,
 996,
 998]
```

Now the same thing but with list comprehension.

In [65]:

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

Out[65]:

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

```
38,
40,
42,
44,
46,
48,
50,
52,
54,
56,
58,
60,
62,
64,
66,
68,
70,
72,
74,
76,
78,
80,
82,
84,
86,
88,
90,
92,
94,
96,
98,
100,
102,
104,
106,
108,
110,
112,
114,
116,
118,
120,
122,
124,
126,
128,
130,
132,
134,
136,
138,
140,
142,
144,
146,
148,
150,
152,
154,
156,
158,
160,
162,
164,
166,
168,
170,
172,
174,
176,
178,
180,
```

```
182,
184,
186,
188,
190,
192,
194,
196,
198,
200,
202,
204,
206,
208,
210,
212,
214,
216,
218,
220,
222,
224,
226,
228,
230,
232,
234,
236,
238,
240,
242,
244,
246,
248,
250,
252,
254,
256,
258,
260,
262,
264,
266,
268,
270,
272,
274,
276,
278,
280,
282,
284,
286,
288,
290,
292,
294,
296,
298,
300,
302,
304,
306,
308,
310,
312,
314,
316,
318,
320,
322,
324,
```

326,
328,
330,
332,
334,
336,
338,
340,
342,
344,
346,
348,
350,
352,
354,
356,
358,
360,
362,
364,
366,
368,
370,
372,
374,
376,
378,
380,
382,
384,
386,
388,
390,
392,
394,
396,
398,
400,
402,
404,
406,
408,
410,
412,
414,
416,
418,
420,
422,
424,
426,
428,
430,
432,
434,
436,
438,
440,
442,
444,
446,
448,
450,
452,
454,
456,
458,
460,
462,
464,
466,
468,

```
470,
472,
474,
476,
478,
480,
482,
484,
486,
488,
490,
492,
494,
496,
498,
500,
502,
504,
506,
508,
510,
512,
514,
516,
518,
520,
522,
524,
526,
528,
530,
532,
534,
536,
538,
540,
542,
544,
546,
548,
550,
552,
554,
556,
558,
560,
562,
564,
566,
568,
570,
572,
574,
576,
578,
580,
582,
584,
586,
588,
590,
592,
594,
596,
598,
600,
602,
604,
606,
608,
610,
612,
```

614,
616,
618,
620,
622,
624,
626,
628,
630,
632,
634,
636,
638,
640,
642,
644,
646,
648,
650,
652,
654,
656,
658,
660,
662,
664,
666,
668,
670,
672,
674,
676,
678,
680,
682,
684,
686,
688,
690,
692,
694,
696,
698,
700,
702,
704,
706,
708,
710,
712,
714,
716,
718,
720,
722,
724,
726,
728,
730,
732,
734,
736,
738,
740,
742,
744,
746,
748,
750,
752,
754,
756,

758,
760,
762,
764,
766,
768,
770,
772,
774,
776,
778,
780,
782,
784,
786,
788,
790,
792,
794,
796,
798,
800,
802,
804,
806,
808,
810,
812,
814,
816,
818,
820,
822,
824,
826,
828,
830,
832,
834,
836,
838,
840,
842,
844,
846,
848,
850,
852,
854,
856,
858,
860,
862,
864,
866,
868,
870,
872,
874,
876,
878,
880,
882,
884,
886,
888,
890,
892,
894,
896,
898,
900,

```
 902,
 904,
 906,
 908,
 910,
 912,
 914,
 916,
 918,
 920,
 922,
 924,
 926,
 928,
 930,
 932,
 934,
 936,
 938,
 940,
 942,
 944,
 946,
 948,
 950,
 952,
 954,
 956,
 958,
 960,
 962,
 964,
 966,
 968,
 970,
 972,
 974,
 976,
 978,
 980,
 982,
 984,
 986,
 988,
 990,
 992,
 994,
 996,
 998]
```

The Python Programming Language: Numerical Python (NumPy)

In [60]:

```
import numpy as np
```

Creating Arrays

Create a list and convert it to a numpy array

In [61]:

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

Out[61]:

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

Or just pass in a list directly

In [62]:

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

Out[62]:

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

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

In [63]:

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

Out[63]:

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

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

In [64]:

```
m.shape
```

Out[64]:

```
(2, 3)
```

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

In [67]:

```
n = np.arange(0, 100, 5) # start at 0 count up by 2, stop before 30
n.shape
```

Out[67]:

```
(20,)
```

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

In [68]:

```
n = n.reshape(4, 5) # reshape array to be 3x5
n
```

Out[68]:

```
array([[ 0,  5, 10, 15, 20],
       [25, 30, 35, 40, 45],
       [50, 55, 60, 65, 70],
       [75, 80, 85, 90, 95]])
```

`linspace` **returns evenly spaced numbers over a specified interval.**

In [70]:

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

Out[70]:

```
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 [74]:

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

Out[74]:

```
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 [72]:

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

Out[72]:

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

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

In [73]:

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

Out[73]:

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

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

In [74]:

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

Out[74]:

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

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

In [75]:

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

Out[75]:

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

Repeat elements of an array using `repeat`.

In [76]:

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

Out[76]:

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

Combining Arrays

In [77]:

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

Out[77]:

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

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

In [82]:

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

Out[82]:

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

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

In [83]:

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

Out[83]:

```
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 [78]:

```
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 [79]:

```
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 [80]:

```
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 [87]:

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

Out[87]:

```
32
```

In [88]:

```
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 [81]:

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

```
z
```

Out[81]:

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

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

In [82]:

```
z.shape
```

Out[82]:

```
(2, 3)
```

Use `.T` to get the transpose.

In [83]:

```
z.T
```

Out[83]:

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

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

In [84]:

```
z.T.shape
```

Out[84]:

```
(3, 2)
```

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

In [85]:

```
z.dtype
```

Out[85]:

```
dtype('int32')
```

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

In [87]:

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

Out[87]:

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

Math Functions

Math Functions

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

In [88]:

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

In [89]:

```
a.sum()
```

Out[89]:

```
3
```

In [90]:

```
a.max()
```

Out[90]:

```
5
```

In [91]:

```
a.min()
```

Out[91]:

```
-4
```

In [92]:

```
a.mean()
```

Out[92]:

```
0.6
```

In [93]:

```
a.std()
```

Out[93]:

```
3.2619012860600183
```

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

In [94]:

```
a.argmax()
```

Out[94]:

```
4
```

In [95]:

```
a.argmin()
```

Out[95]:

```
0
```

In [98]:

```
a.cumsum()
```

Out[98]:

```
array([-4, -6, -5, -2,  3], dtype=int32)
```

array([-4, -3, -2, -1, 0], dtype=int32)

Indexing / Slicing

In [99]:

```
s = np.arange(13)**2
s
```

Out[99]:

```
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 [100]:

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

Out[100]:

```
(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 [101]:

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

Out[101]:

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

Use negatives to count from the back.

In [102]:

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

Out[102]:

```
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 [103]:

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

Out[103]:

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

Let's look at a multidimensional array.

In [107]:

```
r = np.arange(36)
r.resize((6, 6))
r
```

Out[107]:

```
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]])
```

In [108]:

```
r.resize((3,4,3))
print(r)
```

```
[[[ 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]]]
```

In [105]:

```
r[2, 2]
```

Out[105]:

```
14
```

And use : to select a range of rows or columns

In [106]:

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

Out[106]:

```
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 [112]:

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

Out[112]:

```
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 [113]:

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

Out[113]:

```
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 [114]:

```
r[r > 30]
```

Out[114]:

```
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 [115]:

```
r[r > 30] = 30
r
```

Out[115]:

```
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 [116]:

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

Out[116]:

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

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

In [117]:

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

Out[117]:

```
array([[0, 0, 0],
       [0, 0, 0],
```

```
       [0, 0, 0]])
```

Use bracket notation to slice: `array[row, column]`

`r` has also been changed!

In [118]:

```
r
```

Out[118]:

```
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 [119]:

```
r_copy = r.copy()
r_copy
```

Out[119]:

```
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 [120]:

```
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 [121]:

```
test = np.random.randint(0, 10, (4,3))
test
```

Out[121]:

```
array([[1, 0, 5],
       [8, 7, 0],
       [8, 0, 6],
       [6, 6, 7]])
```

Iterate by row:

In [122]:

```
for row in test:
    print(row)
```

```
[1 0 5]
[8 7 0]
[8 0 6]
[6 6 7]
```

Iterate by index:

In [123]:

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

```
[1 0 5]
[8 7 0]
[8 0 6]
[6 6 7]
```

Iterate by row and index:

In [124]:

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

```
row 0 is [1 0 5]
row 1 is [8 7 0]
row 2 is [8 0 6]
row 3 is [6 6 7]
```

Use `zip` to iterate over multiple iterables.

In [125]:

```
test2 = test**2
test2
```

Out[125]:

```
array([[ 1,  0, 25],
       [64, 49,  0],
       [64,  0, 36],
       [36, 36, 49]], dtype=int32)
```

In [126]:

```
for i, j in zip(test, test2):
```

```
    print(i,'+',j,'=',i+j)
```

```
[1 0 5] + [ 1  0 25] = [ 2  0 30]
[8 7 0] + [64 49  0] = [72 56  0]
[8 0 6] + [64  0 36] = [72  0 42]
[6 6 7] + [36 36 49] = [42 42 56]
```

In [ ]:

```
```