Week-1

July 3, 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.

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

add_numbers(1, 2)
```

[1]:
```
3
```

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

[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
```

`add_numbers` updated to take an optional flag parameter.

[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`.

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

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

[4]:
```
3
```

The Python Programming Language: Types and Sequences

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

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

[5]:
```
str
```

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

[6]:
```
NoneType
```

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

[7]:
```
int
```

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

[8]:
```
float
```

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

[9]:
```
function
```

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

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

[10]:
```
tuple
```

Lists are a mutable data structure.

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

[11]:
```
list
```

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

[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.

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

```
1
a
2
b
3.3
```

Or using the indexing operator:

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

```
1
a
2
b
3.3
```

Use + to concatenate lists.

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

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

Use * to repeat lists.

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

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

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

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

[17]:
```
True
```

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

[18]:
```
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.

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

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

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

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

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

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

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

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

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

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

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

[23]:
```
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.

[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.

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

```
␣
↪---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-25-1623ac76de6e> in <module>()
----> 1 'Chris' + 2

TypeError: must be str, not int
```

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

Dictionaries associate keys with values.

[26]:
```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.com'}
x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

[26]:
```
'brooksch@umich.edu'
```

[27]:
```
x['Kevyn Collins-Thompson'] = None
x['Kevyn Collins-Thompson']
```

Iterate over all of the keys:

[28]:
```
for name in x:
    print(x[name])
```

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

Iterate over all of the values:

[29]:
```
for email in x.values():
    print(email)
```

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

Iterate over all of the items in the list:

[30]:
```
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:

[31]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

[32]:
```
fname
```

[32]:
```
'Christopher'
```

[33]:
```
lname
```

[33]:
```
'Brooks'
```

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

[34]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email = x
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-34-9ce70064f53e> 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

[35]:
```
print('Chris' + 2)
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-35-82ccfdd3d5d3> in <module>()
----> 1 print('Chris' + 2)


TypeError: must be str, not int
```

[36]:
```
print('Chris' + str(2))
```

```
Chris2
```

Python has a built in method for convenient string formatting.

[37]:
```
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

[38]:
```
import csv

%precision 2

with open('mpg.csv') as csvfile:
    mpg = list(csv.DictReader(csvfile))

mpg[:3] # The first three dictionaries in our list.
```

[38]:
```
[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.

[39]:
```
len(mpg)
```

[39]:
```
234
```

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

[40]:
```
mpg[0].keys()
```

[40]:
```
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.

[41]:
```
sum(float(d['cty']) for d in mpg) / len(mpg)
```

[41]:
```
16.86
```

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

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

[42]:
```
23.44
```

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

[43]:
```
cylinders = set(d['cyl'] for d in mpg)
cylinders
```

[43]:
```
{'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.

[44]:
```
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
```

[44]:
```
[('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.

[45]:
```
vehicleclass = set(d['class'] for d in mpg) # what are the class types
vehicleclass
```

[45]:
```
{'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.

[46]:
```
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
```

[46]:
```
[('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

[47]:
```
import datetime as dt
import time as tm
```

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

[48]:
```
tm.time()
```

[48]:
```
1593738228.76
```

Convert the timestamp to datetime.

[49]:
```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

[49]:
```
datetime.datetime(2020, 7, 3, 1, 3, 48, 813875)
```

Handy datetime attributes:

[50]:
```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year, month, day, etc.from a datetime
```

[50]:
```
(2020, 7, 3, 1, 3, 48)
```

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

[51]:
```
delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

[51]:
```
datetime.timedelta(100)
```

`date.today` returns the current local date.

[52]:
```
today = dt.date.today()
```

[53]:
```
today - delta # the date 100 days ago
```

[53]:
```
datetime.date(2020, 3, 25)
```

[54]:
```
today > today-delta # compare dates
```

[54]:
```
True
```

The Python Programming Language: Objects and map()

An example of a class in python:

[55]:
```
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
```

[56]:
```
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.

[57]:
```
store1 = [10.00, 11.00, 12.34, 2.34]
store2 = [9.00, 11.10, 12.34, 2.01]
cheapest = map(min, store1, store2)
cheapest
```

[57]:
```
<map at 0x7f2abacb5dd8>
```

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

[58]:
```
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.

[59]:
```
my_function = lambda a, b, c : a + b
```

[60]:
```
my_function(1, 2, 3)
```

[60]:
```
3
```

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

[61]:
```
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list
```

[61]:
```
[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]
```

Now the same thing but with list comprehension.

[62]:
```
my_list = [number for number in range(0,1000) if number % 2 == 0]
my_list
```

[62]:
```
[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)

[63]:
```
import numpy as np
```

Creating Arrays

Create a list and convert it to a numpy array

[64]:
```
mylist = [1, 2, 3]
x = np.array(mylist)
x
```

[64]:
```
array([1, 2, 3])
```

Or just pass in a list directly

[65]:
```
y = np.array([4, 5, 6])
y
```

[65]:
```
array([4, 5, 6])
```

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

[66]:
```
m = np.array([[7, 8, 9], [10, 11, 12]])
m
```

[66]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

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

[67]:
```
m.shape
```

[67]:
```
(2, 3)
```

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

[68]:
```
n = np.arange(0, 30, 2) # start at 0 count up by 2, stop before 30
n
```

[68]:
```
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.

[69]:
```
n = n.reshape(3, 5) # reshape array to be 3x5
n
```

[69]:
```
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.

[70]:
```
o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
o
```

[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.

[71]:
```
o.resize(3, 3)
o
```

[71]:
```
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.

[72]:
```
np.ones((3, 2))
```

[72]:
```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

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

[73]:
```
np.zeros((2, 3))
```

[73]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

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

[74]:
```
np.eye(3)
```

[74]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

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

[75]:
```
np.diag(y)
```

[75]:
```
array([[4, 0, 0],
       [0, 5, 0],
       [0, 0, 6]])
```

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

[76]:
```
np.array([1, 2, 3] * 3)
```

[76]:
```
array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Repeat elements of an array using `repeat`.

[77]:
```
np.repeat([1, 2, 3], 3)
```

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

Combining Arrays

[78]:
```
p = np.ones([2, 3], int)
p
```

[78]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

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

[79]:
```
np.vstack([p, 2*p])
```

[79]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [2, 2, 2],
       [2, 2, 2]])
```

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

[80]:
```
np.hstack([p, 2*p])
```

[80]:
```
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.

[81]:
```
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]
```

[82]:
```
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 ]
```

[83]:
```
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_1y_1 + x_2y_2 + x_3y_3$

[84]:
```
x.dot(y) # dot product  1*4 + 2*5 + 3*6
```

[84]:
```
32
```

[85]:
```
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.

[86]:
```
z = np.array([y, y**2])
z
```

[86]:
```
array([[ 4,  5,  6],
       [16, 25, 36]])
```

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

[87]:
```
z.shape
```

[87]:
```
(2, 3)
```

Use `.T` to get the transpose.

[88]:
```
z.T
```

[88]:
```
array([[ 4, 16],
       [ 5, 25],
       [ 6, 36]])
```

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

[89]:
```
z.T.shape
```

[89]:
```
(3, 2)
```

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

[90]:
```
z.dtype
```

[90]:
```
dtype('int64')
```

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

[91]:
```
z = z.astype('f')
z.dtype
```

[91]:
```
dtype('float32')
```

Math Functions

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

[92]:
```
a = np.array([-4, -2, 1, 3, 5])
```

[93]:
```
a.sum()
```

[93]:
```
3
```

[94]:
```
a.max()
```

[94]:
```
5
```

[95]:
```
a.min()
```

[95]:
```
-4
```

[96]:
```
a.mean()
```

[96]:
```
0.60
```

[97]:
```
a.std()
```

[97]:
```
3.26
```

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

[98]:
```
a.argmax()
```

[98]:
```
4
```

[99]:
```
a.argmin()
```

[99]:
```
0
```

Indexing / Slicing

[100]:
```
s = np.arange(13)**2
s
```

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

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

[101]:
```
s[0], s[4], s[-1]
```

[101]:
```
(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.

[102]:
```
s[1:5]
```

[102]:
```
array([ 1,  4,  9, 16])
```

Use negatives to count from the back.

[103]:
```
s[-4:]
```

[103]:
```
array([ 81, 100, 121, 144])
```

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.

[104]:
```
s[-5::-2]
```

[104]:
```
array([64, 36, 16,  4,  0])
```

Let's look at a multidimensional array.

[105]:
```
r = np.arange(36)
r.resize((6, 6))
r
```

[105]:
```
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]`

[106]:
```
r[2, 2]
```

[106]:
```
14
```

And use : to select a range of rows or columns

[107]:
```
r[3, 3:6]
```

[107]:
```
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.

[108]:
```
r[:2, :-1]
```

[108]:
```
array([[ 0,  1,  2,  3,  4],
       [ 6,  7,  8,  9, 10]])
```

This is a slice of the last row, and only every other element.

[109]:
```
r[-1, ::2]
```

[109]:
```
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`)

[110]:
```
r[r > 30]
```

[110]:
```
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.

[111]:
```
r[r > 30] = 30
r
```

[111]:
```
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`

[112]:
```
r2 = r[:3,:3]
r2
```

[112]:
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [12, 13, 14]])
```

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

[113]:
```
r2[:] = 0
r2
```

[113]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

r has also been changed!

[114]:
```
r
```

[114]:
```
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

[115]:
```
r_copy = r.copy()
r_copy
```

[115]:
```
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.

[116]:
```
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.

[117]:
```
test = np.random.randint(0, 10, (4,3))
test
```

[117]:
```
array([[4, 0, 1],
       [6, 0, 1],
       [5, 0, 4],
       [8, 2, 2]])
```

Iterate by row:

[118]:
```
for row in test:
    print(row)
```

```
[4 0 1]
[6 0 1]
[5 0 4]
[8 2 2]
```

Iterate by index:

[119]:
```
for i in range(len(test)):
    print(test[i])
```

```
[4 0 1]
[6 0 1]
[5 0 4]
[8 2 2]
```

Iterate by row and index:

[120]:
```
for i, row in enumerate(test):
    print('row', i, 'is', row)
```

```
row 0 is [4 0 1]
row 1 is [6 0 1]
row 2 is [5 0 4]
row 3 is [8 2 2]
```

Use `zip` to iterate over multiple iterables.

[121]:
```
test2 = test**2
test2
```

[121]:
```
array([[16,  0,  1],
       [36,  0,  1],
       [25,  0, 16],
       [64,  4,  4]])
```

[122]:
```
for i, j in zip(test, test2):
    print(i,'+',j,'=',i+j)
```

```
[4 0 1] + [16  0  1] = [20  0  2]
[6 0 1] + [36  0  1] = [42  0  2]
[5 0 4] + [25  0 16] = [30  0 20]
[8 2 2] + [64  4  4] = [72  6  6]
```