

# Week 1

January 8, 2020

---

*You are currently looking at **version 1.1** of this notebook. To download notebooks and datafiles, as well as get help on Jupyter notebooks in the Coursera platform, visit the [Jupyter Notebook FAQ](#) course resource.*

---

## 1 The Python Programming Language: Functions

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

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

```
        add_numbers(1, 2)
```

```
Out[1]: 3
```

`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
```

`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
```

# The Python Programming Language: Types and Sequences  
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
```

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

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

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

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 character
         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
```

split returns a list of all the words in a string, or a list split on a specific character.

```
In [24]: firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the first element
        lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the last element
        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-25-1623ac76de6e> in <module>()
----> 1 'Chris' + 2

TypeError: must be str, not int
```

```
In [26]: 'Chris' + str(2)
```

```
Out[26]: 'Chris2'
```

Dictionaries associate keys with values.

```
In [27]: x = {'Christopher Brooks': 'broosch@umich.edu', 'Bill Gates': 'billg@microsoft.com'}
        x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

```
Out[27]: 'broosch@umich.edu'
```

```
In [28]: x['Kevyn Collins-Thompson'] = None
        x['Kevyn Collins-Thompson']
```

Iterate over all of the keys:

```
In [29]: for name in x:
        print(x[name])
```

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

Iterate over all of the values:

```
In [30]: for email in x.values():
         print(email)
```

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

Iterate over all of the items in the list:

```
In [31]: for name, email in x.items():
         print(name)
         print(email)
```

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

You can unpack a sequence into different variables:

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

```
In [33]: fname
```

```
Out[33]: 'Christopher'
```

```
In [34]: lname
```

```
Out[34]: 'Brooks'
```

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

```
In [35]: x = ('Christopher', 'Brooks', 'brooks@umich.edu', 'Ann Arbor')
         fname, lname, email = x
```

```
-----  
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-35-9ce70064f53e> in <module>()  
    1 x = ('Christopher', 'Brooks', 'brooks@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 [36]: print('Chris' + 2)
```

```
-----  
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-36-82ccfdd3d5d3> in <module>()  
----> 1 print('Chris' + 2)
```

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

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

```
Chris2
```

Python has a built in method for convenient string formatting.

```
In [38]: 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 [39]: 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[39]: [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 [40]: len(mpg)
```

```
Out[40]: 234
```

keys gives us the column names of our csv.

```
In [41]: mpg[0].keys()
```

```
Out[41]: odict_keys(['', 'manufacturer', 'model', 'displ', 'year', 'cyl', 'trans', 'drv', 'cty',
```

This is how to find the average cty fuel economy across all cars. All values in the dictionaries are strings, so we need to convert to float.

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

```
Out[42]: 16.86
```

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

```
In [43]: sum(float(d['hwy']) for d in mpg) / len(mpg)
```

```
Out[43]: 23.44
```

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

```
In [44]: cylinders = set(d['cyl'] for d in mpg)
cylinders
```

```
Out[44]: {'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 [45]: 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[45]: [('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 [46]: vehicleclass = set(d['class'] for d in mpg) # what are the class types
vehicleclass
```

```
Out[46]: {'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 [47]: 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 hwy mpg')

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

```
Out[47]: [('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 [48]: import datetime as dt
import time as tm
```

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

```
In [49]: tm.time()
```

```
Out[49]: 1578470767.22
```

Convert the timestamp to datetime.

```
In [50]: dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

```
Out[50]: datetime.datetime(2020, 1, 8, 8, 6, 20, 297752)
```

Handy datetime attributes:

```
In [51]: dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get year,
```

```
Out[51]: (2020, 1, 8, 8, 6, 20)
```

timedelta is a duration expressing the difference between two dates.

```
In [52]: delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

```
Out[52]: datetime.timedelta(100)
```

date.today returns the current local date.

```
In [53]: today = dt.date.today()
```

```
In [54]: today - delta # the date 100 days ago
```

```
Out[54]: datetime.date(2019, 9, 30)
```

```
In [55]: today > today-delta # compare dates
```

```
Out[55]: True
```

# The Python Programming Language: Objects and map()  
An example of a class in python:

```
In [56]: 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 [57]: 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 [58]: 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[58]: <map at 0x7f1370cfc4e0>
```

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

```
In [59]: 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 [62]: my_function = lambda a, b, c : a + b
```

```
In [63]: my_function(1, 2, 3)
```

```
Out[63]: 3
```

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

```
In [64]: my_list = []
        for number in range(0, 1000):
            if number % 2 == 0:
                my_list.append(number)
        my_list
```

```
Out[64]: [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.

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

## Creating Arrays

Create a list and convert it to a numpy array

```
In [67]: mylist = [1, 2, 3]
         x = np.array(mylist)
         x
```

```
Out[67]: array([1, 2, 3])
```

Or just pass in a list directly

```
In [68]: y = np.array([4, 5, 6])
         y
```

```
Out[68]: array([4, 5, 6])
```

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

```
In [69]: m = np.array([[7, 8, 9], [10, 11, 12]])
         m
```

```
Out[69]: array([[ 7,  8,  9],
                [10, 11, 12]])
```

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

```
In [70]: m.shape
```

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

arange returns evenly spaced values within a given interval.

```
In [71]: n = np.arange(0, 30, 2) # start at 0 count up by 2, stop before 30
         n
```

```
Out[71]: 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 [72]: n = n.reshape(3, 5) # reshape array to be 3x5
         n
```

```
Out[72]: 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 [73]: o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
         o
```

```
Out[73]: 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)
         0
```

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

```
Out[75]: array([[ 1.,  1.],
                [ 1.,  1.],
                [ 1.,  1.]])
```

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

```
In [76]: np.zeros((2, 3))
```

```
Out[76]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])
```

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

```
In [77]: np.eye(3)
```

```
Out[77]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

diag extracts a diagonal or constructs a diagonal array.

```
In [78]: np.diag(y)
```

```
Out[78]: array([[4, 0, 0],
                [0, 5, 0],
                [0, 0, 6]])
```

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

```
In [79]: np.array([1, 2, 3] * 3)
```

```
Out[79]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Repeat elements of an array using `repeat`.

```
In [80]: np.repeat([1, 2, 3], 3)
```

```
Out[80]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

#### Combining Arrays