*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) course resource.*

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 [4]:
```
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))
print(add_numbers(1, 2, 3,4))
```

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

Out[11]:
```
tuple
```

Lists are a mutable data structure.

In [15]:
```
x = [1, 'a', 2, 'b']
type(x)
```

Out[15]:
```
list
```

Use append to append an object to a list.

In [16]:
```
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 [17]:
```
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 [18]:
```
[1,2] + [3,4]
```

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

Use * to repeat lists.

In [21]:
```
[1]*3
```

Out[21]:
```
[1, 1, 1]
```

In [22]:
```
[1,2]*3
```

Out[22]:
```
[1, 2, 1, 2, 1, 2]
```

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 [25]:
```
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
print(x[0:0]) #print nothing
```

```
T
T
Th
```

This will return the last element of the string.

In [26]:
```
x[-1]
```

Out[26]:
```
'g'
```

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

In [27]:
```
x[-4:-2]
```

Out[27]:
```
'ri'
```

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

In [28]:
```
x[:3]
```

Out[28]:
```
'Thi'
```

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

In [29]:
```
x[3:]
```

Out[29]:
```
's is a string'
```

In [31]:
```
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 [32]:
```
firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0] # [0] selects the fi
lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1] # [-1] selects the l
print(firstname)
print(lastname)
```

```
Christopher
Brooks
```

Make sure you convert objects to strings before concatenating.

In [33]:
```
'Chris' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-1623ac76de6e> in <module>()
----> 1 'Chris' + 2

TypeError: must be str, not int
```

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

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

Maps/Dictionaries associate keys with values.

In [35]:
```
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.c
x['Christopher Brooks'] # Retrieve a value by using the indexing operator
```

Out[35]:
```
'brooksch@umich.edu'
```

In [39]:
```
x['Kevyn Collins-Thompson'] = None
print(x)
x['Kevyn Collins-Thompson']
```

```
{'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.co
m', 'Kevyn Collins-Thompson': None}
```

Iterate over all of the keys:

In [40]:
```
for name in x:
    print(x[name])
```

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

Iterate over all of the values:

In [41]:
```
for email in x.values():
    print(email)
```

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

Iterate over all of the items in the list:

In [42]:
```
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 [43]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
```

In [44]:
```
fname
```

Out[44]:
```
'Christopher'
```

In [45]:
```
lname
```

Out[45]:
```
'Brooks'
```

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

In [47]:
```
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email = x
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-47-9ce70064f53e> 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

In [48]:
```
print('Chris' + 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-82ccfdd3d5d3> in <module>()
----> 1 print('Chris' + 2)

TypeError: must be str, not int
```

In [49]:
```
print('Chris' + str(2))
```

```
Chris2
```

Python has a built in method for convenient string formatting.

In [51]:
```
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 [52]:
```
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[52]:
```
[OrderedDict([('', '1'),
              ('manufacturer', 'audi'),
              ('model', 'a4'),
              ('displ', '1.8'),
              ('year', '1999'),
              ('cyl', '4'),
              ('trans', 'auto(l5)'),
              ('drv', 'f'),
              ('cty', '18'),
              ('hwy', '29'),
              ('fl', 'p'),
              ('class', 'compact')]),
 OrderedDict([('', '2'),
              ('manufacturer', 'audi'),
              ('model', 'a4'),
              ('displ', '1.8'),
              ('year', '1999'),
              ('cyl', '4'),
              ('trans', 'manual(m5)'),
```

`csv.Dictreader` has read in each row of our csv file as a dictionary. `len` shows that our list is comprised of 234 dictionaries.

In [53]:
```
len(mpg)
```

Out[53]:
```
234
```

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

In [54]:
```
mpg[0].keys()
```

Out[54]:
```
odict_keys(['', 'manufacturer', 'model', 'displ', 'year', 'cyl', 'trans', 'dr
v', 'cty', 'hwy', 'fl', 'class'])
```

In [56]:
```
mpg[0].values()
```

Out[56]:
```
odict_values(['1', 'audi', 'a4', '1.8', '1999', '4', 'auto(l5)', 'f', '18', '2
9', 'p', 'compact'])
```

In [57]:
```
mpg[0].items()
```

Out[57]:
```
odict_items([('', '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')])
```

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

In [73]:
```
sum(float(d['cty']) for d in mpg) / len(mpg) # list comprehensions
```

Out[73]:
```
16.86
```

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

In [74]:
```
sum(float(d['hwy']) for d in mpg) / len(mpg)
```

Out[74]:
```
23.44
```

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

In [75]:
```
cylinders = set(d['cyl'] for d in mpg)
cylinders
```

Out[75]:
```
{'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 [76]:
```
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'

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

Out[76]:
```
[('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 [77]:
```
vehicleclass = set(d['class'] for d in mpg) # what are the class types
vehicleclass
```

Out[77]:
```
{'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 [78]:
```
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',

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

Out[78]:
```
[('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 [79]:
```
import datetime as dt
import time as tm
```

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

In [80]:
```
tm.time()
```

Out[80]:
```
1542513883.68
```

Convert the timestamp to datetime.

In [81]:
```
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
```

Out[81]:
```
datetime.datetime(2018, 11, 18, 4, 4, 46, 269087)
```

Handy datetime attributes:

In [82]:
```
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second # get
```

Out[82]:
```
(2018, 11, 18, 4, 4, 46)
```

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

In [83]:
```
delta = dt.timedelta(days = 100) # create a timedelta of 100 days
delta
```

Out[83]:
```
datetime.timedelta(100)
```

`date.today` returns the current local date.

In [84]:
```
today = dt.date.today()
```

In [85]:
```
today - delta # the date 100 days ago
```

Out[85]:
```
datetime.date(2018, 8, 10)
```

In [86]:
```
today > today-delta # compare dates
```

Out[86]:
```
True
```

The Python Programming Language: Objects and map()

An example of a class in python:

In [87]:
```
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 [88]:
```
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.l
```

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

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

In [94]:
```
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[94]:
```
<map at 0x7f3ed012cba8>
```

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

In [95]:
```
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 [96]:
```
my_function = lambda a, b, c : a + b
```

In [97]:
```
my_function(1, 2, 3)
```

Out[97]:
```
3
```

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

In [98]:
```
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list
```

Out[98]:
```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
```

Now the same thing but with list comprehension.

In [99]:
```
my_list = [number for number in range(0,1000) if number % 2 == 0]
my_list
```

Out[99]:
```
[0,
 2,
 4,
 6,
 8,
 10,
 12,
 14,
 16,
 18,
 20,
 22,
 24,
 26,
 28,
 30,
 32,
 34,
 36,
```

The Python Programming Language: Numerical Python (NumPy)

In [100]:
```
import numpy as np
```

Creating Arrays

Create a list and convert it to a numpy array

In [101]:
```
mylist = [1, 2, 3]
x = np.array(mylist)
x
```

Out[101]:
```
array([1, 2, 3])
```

Or just pass in a list directly

In [102]:
```
y = np.array([4, 5, 6])
y
```

Out[102]:
```
array([4, 5, 6])
```

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

In [103]:
```
m = np.array([[7, 8, 9], [10, 11, 12]])
m
```

Out[103]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

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

In [104]:
```
m.shape
```

Out[104]:
```
(2, 3)
```

arange returns evenly spaced values within a given interval.

In [105]:
```
n = np.arange(0, 30, 2) # start at 0 count up by 2, stop before 30
n
```

Out[105]:
```
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 [106]:
```
n = n.reshape(3, 5) # reshape array to be 3x5
n
```

Out[106]:
```
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 [107]:
```
o = np.linspace(0, 4, 9) # return 9 evenly spaced values from 0 to 4
o
```

Out[107]:
```
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 [108]:
```
o.resize(3, 3)
o
```

Out[108]:
```
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 [109]:
```
np.ones((3, 2))
```

Out[109]:
```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

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

In [110]:
```
np.zeros((2, 3))
```

Out[110]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

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

In [111]:
```
np.eye(3)
```

Out[111]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

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

In [112]:
```
np.diag(y)
```

Out[112]:
```
array([[4, 0, 0],
       [0, 5, 0],
       [0, 0, 6]])
```

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

In [113]:
```
np.array([1, 2, 3] * 3)
```

Out[113]:
```
array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

Repeat elements of an array using `repeat`.

In [114]:
```
np.repeat([1, 2, 3], 3)
```

Out[114]:
```
array([1, 1, 1, 2, 2, 2, 3, 3, 3])
```

Combining Arrays

In [115]:
```
p = np.ones([2, 3], int)
p
```

Out[115]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

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

In [116]:
```
np.vstack([p, 2*p])
```

Out[116]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [2, 2, 2],
       [2, 2, 2]])
```

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

In [117]:
```
np.hstack([p, 2*p])
```

Out[117]:
```
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 [118]:
```
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 [119]:
```
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 [120]:
```
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 [121]:
```
x.dot(y) # dot product  1*4 + 2*5 + 3*6
```

Out[121]:
```
32
```

In [122]:
```
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 [123]:
```
z = np.array([y, y**2])
z
```

Out[123]:
```
array([[ 4,  5,  6],
       [16, 25, 36]])
```

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

In [124]:
```
z.shape
```

Out[124]:
```
(2, 3)
```

Use `.T` to get the transpose.

In [125]:
```
z.T
```

Out[125]:
```
array([[ 4, 16],
       [ 5, 25],
       [ 6, 36]])
```

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

In [126]:
```
z.T.shape
```

Out[126]:
```
(3, 2)
```

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

In [127]:
```
z.dtype
```

Out[127]:
```
dtype('int64')
```

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

In [128]:
```
z = z.astype('f')
z.dtype
```

Out[128]:
```
dtype('float32')
```

Math Functions

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

In [129]:
```
a = np.array([-4, -2, 1, 3, 5])
```

In [130]:
```
a.sum()
```

Out[130]: 3

In [131]:
```
a.max()
```

Out[131]: 5

In [132]:
```
a.min()
```

Out[132]: -4

In [133]:
```
a.mean()
```

Out[133]: 0.60

In [134]:
```
a.std()
```

Out[134]: 3.26

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

In [135]:
```
a.argmax()
```

Out[135]: 4

In [136]:
```
a.argmin()
```

Out[136]: 0

Indexing / Slicing

In [137]:
```
s = np.arange(13)**2
s
```

Out[137]: 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.

In [138]:
```
s[0], s[4], s[-1]
```

Out[138]: (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 [139]:
```
s[1:5]
```

Out[139]:
```
array([ 1,  4,  9, 16])
```

Use negatives to count from the back.

In [140]:
```
s[-4:]
```

Out[140]:
```
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.

In [141]:
```
s[-5::-2]
```

Out[141]:
```
array([64, 36, 16,  4,  0])
```

Let's look at a multidimensional array.

In [142]:
```
r = np.arange(36)
r.resize((6, 6))
r
```

Out[142]:
```
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 [143]:
```
r[2, 2]
```

Out[143]:
```
14
```

And use : to select a range of rows or columns

In [144]:
```
r[3, 3:6]
```

Out[144]:
```
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 [145]:
```
r[:2, :-1]
```

Out[145]:
```
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 [146]:
```
r[-1, ::2]
```

Out[146]:
```
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 [147]:
```
r[r > 30]
```

Out[147]:
```
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 [148]:
```
r[r > 30] = 30
r
```

Out[148]:
```
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 [149]:
```
r2 = r[:3,:3]
r2
```

Out[149]:
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [12, 13, 14]])
```

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

In [150]:
```
r2[:] = 0
r2
```

Out[150]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

`r` has also been changed!

In [151]:
```
r
```

Out[151]:
```
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 [152]:
```
r_copy = r.copy()
r_copy
```

Out[152]:
```
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 [153]:
```
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 [154]:
```
test = np.random.randint(0, 10, (4,3))
test
```

Out[154]:
```
array([[8, 1, 1],
       [9, 4, 9],
       [4, 0, 8],
       [1, 9, 6]])
```

Iterate by row:

In [155]:
```
for row in test:
    print(row)
```

```
[8 1 1]
[9 4 9]
[4 0 8]
[1 9 6]
```

Iterate by index:

In [156]:
```
for i in range(len(test)):
    print(test[i])
```

```
[8 1 1]
[9 4 9]
[4 0 8]
[1 9 6]
```

Iterate by row and index:

In [157]:
```
for i, row in enumerate(test):
    print('row', i, 'is', row)
```

```
row 0 is [8 1 1]
row 1 is [9 4 9]
row 2 is [4 0 8]
row 3 is [1 9 6]
```

Use `zip` to iterate over multiple iterables.

In [158]:
```
test2 = test**2
test2
```

Out[158]:
```
array([[64,  1,  1],
       [81, 16, 81],
       [16,  0, 64],
       [ 1, 81, 36]])
```

In [159]:
```
for i, j in zip(test, test2):
    print(i,'+',j,'=',i+j)
```

```
[8 1 1] + [64  1  1] = [72  2  2]
[9 4 9] + [81 16 81] = [90 20 90]
[4 0 8] + [16  0 64] = [20  0 72]
[1 9 6] + [ 1 81 36] = [ 2 90 42]
```