```
from google.colab import drive
drive.mount('/content/drive')

     Mounted at /content/drive
```

L1: Dictionaries and tuples, functions, Numpy, Mathplotlib

This session we introduce Python language essentials with two more compound data structures - tuples and dictionaries - and with writing reusable code blocks (functions). We also introduce the key libraries for handling tabular data: Numpy (short for numerical Python).

1. Tuples

Recall there are four main compound data structures: lists, tuples, sets and dictionaries. Lists are the go-to structure most of the time. They are ordered, mutable collections of elements.

Tuples are immutable ordered collections. They're commonly used to pass data around within programs.

```
# define a tuple like this:

I_am_tuple = (4,5)
I_am_also_a_tuple = ('apex','legend')


# tuples can have more than 2 elements:
mega_tuple = (4,5,6,7,8)


# you can reference within them:
mega_tuple[0]

     4


# but you can't change their elements
mega_tuple[0] = 7

     ---------------------------------------------------------------------------
     TypeError                                 Traceback (most recent call last)
     <ipython-input-6-8f62b999d7af> in <module>()
           1 # but you can't change their elements
     ----> 2 mega_tuple[0] = 7

     TypeError: 'tuple' object does not support item assignment
```

SEARCH STACK OVERFLOW

```
# The numerical ordering of tuples can be a useful property, if data is stored in a
# Let's make a small phone book example


Frodo = ('Frodo','+202 569 8745','frodo@baggins_shire.com')
Sam = ('Sam', '+202 456 5646', 'sam@samwise_gamgee.com')


# Let's print just the people's names:
for person in [Frodo, Sam]:
    print(person[0])
```

```
Frodo
Sam
```

```
# Let's print just the people's emails:
for person in [Frodo, Sam]:
    print(person[2])
```

```
frodo@baggins_shire.com
sam@samwise_gamgee.com
```

```
# Let's print out s formatted string:
for person in [Frodo, Sam]:
    print('My name is {}, and you can call me on {} or email me at {}'.format(perso
```

2. Dictionaries

A dictionary is another fundamental data structure. The value of a dictionary is the ('key':'value') organisation, much like a standard dictionary!

```
fellowship = {'hobbit_1':'Frodo',
              'hobbit_2':'Sam',
              'hobbit_3':'Pippin',
              'hobbit_4':'Merry'}


# call the values from the keys
fellowship['hobbit_1']
```

```
'Frodo'
```

```
# we can also generate lists of both the keys:
fellowship.keys()
```

```
dict_keys(['hobbit_1', 'hobbit_2', 'hobbit_3', 'hobbit_4'])
```

```
# ...and the values:
fellowship.values()
```

```
# question: how would you get a list of the values?
```

```
# we could use dictionaries of dictionaries:

Frodo = {'name':'Frodo','cell':'+202 569 8745','email':'frodo@baggins_shire.com'}
Sam = {'name':'Sam','cell':'+202 456 5646','email':'sam@samwise_gamgee.com'}

fellowship_contact_info = {'Frodo':Frodo,
                          'Sam':Sam}
```

```
fellowship_contact_info['Sam']
```

```
    {'cell': '+202 456 5646', 'email': 'sam@samwise_gamgee.com', 'name': 'Sam'}
```

```
fellowship_contact_info['Sam']['cell']
```

```
    '+202 456 5646'
```

```
fellowship_contact_info['Sam']['email']
```

```
    'sam@samwise_gamgee.com'
```

This is actually the structure of a JSON file - which is organized as a dictionary of nested dictionaries!

3. Combine item pairs with zip()

Say you had a column of latitudes, and a column of longitudes. You want a column of coordinate pairs. The zip() function lets you 'zip' two iterables together, giving you tuples.

```
first_list = [1,2,3]
second_list = ['one', 'two', 'three']
```

```
# it gives you a zip item (good for saving memory)
zip(first_list, second_list)
```

```
    <zip at 0x7f31bfe221e0>
```

```
# turn that item into a list
list(zip(first_list, second_list))
```

```
    [(1, 'one'), (2, 'two'), (3, 'three')]
```

4. Defining functions

So you have written some code for a difficult task (eg. solve Fermat's Last Theorem). You may want to do the same task again. You could (a) memorize the code and re-write it each time; (b) copy and paste it; or (c) write a function. A function is a reusable code block. You can pass data into functions (as parameters). Functions can return data to the main program.

```
# define a function

def my_function():
    print("Hi I'm a function")
```

```
# once defined, call it once or many times

my_function()
```

```
    Hi I'm a function
```

```
# pass data into functions

greeting = "Hi people, this is Python session 3"
print(greeting)
```

```
    Hi people, this is Python session 3
```

The `def` statement introduces a function definition. It expects a function name, parentheses, any parameters the function will take, and a colon. The function code block must be indented. The parentheses are always required, when defining or calling a function, even if no parameters are used.

```
# this function expects one parameter

def sound_more_excited(my_string):
    new_string = my_string + '!!'
    print(new_string)
```

Note: functions have an internal name-space (or symbol table). The data passed into `sound_more_excited` will be referred to, within the function, as `my_string`.

```
sound_more_excited(greeting)
```

```
    Hi people, this is Python session 3!!
```

```
# this function will return data to the main program

def sound_really_excited(my_string):
    new_string = my_string.upper() + '!!!'

sound_really_excited(greeting)
```

```
'HI PEOPLE, THIS IS PYTHON SESSION 3!!!'
```

5. Functions with multiple arguments

Functions can take many arguments. You can make them easy to work with by:

- Defining default arguments.
- Providing keyword arguments.

```
# default arguments allow you to call functions with less typing

def ask_permission(prompt, retries = 3, msg = 'Try again >> '):
    while retries > 0:
        user_input = input(prompt)
        if user_input in ['yes','YES','y']:
            return(True)
        elif user_input in ['no','NO','n']:
            return(False)
        else:
            retries = retries - 1
        print(msg)
```

One parameter (`prompt`) is mandatory. Default values will be assumed for the other parameters, unless they are passed.

```
# function call with only the mandatory argument
ask_permission("delete all files? >> ")
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/usr/local/lib/python3.7/dist-packages/ipykernel/kernelbase.py in
_input_request(self, prompt, ident, parent, password)
    728             try:
--> 729                 ident, reply = self.session.recv(self.stdin_socket,
0)
    730             except Exception:
```

6 frames

```
zmq/backend/cython/socket.pyx in zmq.backend.cython.socket.Socket.recv()
```

```
# function call with one optional parameter
ask_permission("delete *all* the files? >> ", 10)
```

```
zmq/backend/cython/socket.pyx in zmq.backend.cython.socket. recv copy()
```

```
# with all optional parameters
ask_permission("delete *all* the files? >> ", 10, msg = 'expected yes or no >> ')
```

```
During handling of the above exception, another exception occurred:
```

Arguments with a default value as also called `keyword arguments`, those without are `positional arguments`. Positional arguments always need to come before keyword arguments.

```
733                 # re-raise KeyboardInterrupt, to truncate traceback
```

```
# Try not to break your function calls like this:

ask_permission(retries = 6, 'Delete all files')
```

```
KeyboardInterrupt:
```

6. NumPy

NumPy, which stands for Numerical Python, is a fundamental package for high performance scientific computing and data analysis.

The NumPy array (ndarray) is a highly efficient way of storing and manipulating numerical data.

```
import numpy as np
from IPython.display import Image
```

Why use Numpy?

Python has built-in number objects (ints, floats) and container objects (lists, tuples). Lists can contain any object types, which are retrieved from elsewhere in memory, with a range of functionality to insert and change items. By contract, Numpy arrays are homogenous (only numbers) and are optimized for speed in operations like matrix algebra.

```
# speed test - regular
```

```
my_range = range(1000)
%timeit [i**2 for i in my_range]
```

```
    1000 loops, best of 5: 252 µs per loop
```

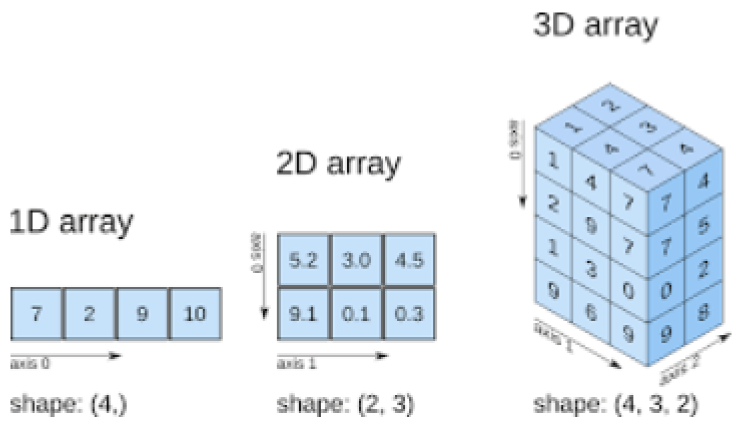
```
# speed test - numpy

my_array = np.arange(1000)
%timeit my_array ** 2
```

```
    The slowest run took 2292.50 times longer than the fastest. This could mean th
    1000000 loops, best of 5: 1.68 µs per loop
```

Create NumPy Arrays:

You can create them manually by passing a list. In reality, you're not usually going to type individual values in. More often, you would:

- (a) load existing data from a file (eg. CSV);
- (b) create arrays based on an arithmetic sequence;
- (c) create a spread of values between a start and end-point;
- (d) or create an empty array to populate later.



```
# create a 1 dimensional array array1
my_list=[1,2,3,4]

array1=np.array(my_list)
array1
```

```
    array([1, 2, 3, 4])
```

```
# create a 2 dimensional array
array2 = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)
array2
```

```
    array([[1, 2, 3, 4],
           [5, 6, 7, 8]])
```

```
# create a 3 dimensional array
array3=np.array([[[1,2,3],[4,5,6]],
                 [[1,2,3],[4,5,6]]])
array3
```

```
array([[[1, 2, 3],
        [4, 5, 6]],

       [[1, 2, 3],
        [4, 5, 6]]])
```

Look up info on the array:

```
my_array=array2
```

```
# number of dimensions
my_array.ndim
```

```
2
```

```
# shape of the array
my_array.shape
```

```
(2, 4)
```

```
# number of elements
my_array.size
```

```
8
```

```
#  memory address
my_array.data
```

```
<memory at 0x7f31c3b04c90>
```

```
#data type
my_array.dtype
```

```
dtype('int64')
```

```
# Change the data type to float
my_array.astype(float).dtype
```

```
dtype('float64')
```

Math operations are performed element-wise

For example `+10`, `*10` or `== 10` would be performed on *each* element in the array.

```
my_array = np.arange(6)
my_array

    array([0, 1, 2, 3, 4, 5])
```

```
#Add code
```

Summarize and compute reductions

The `ndarray` objects have some pretty useful methods

```
array2 = np.arange(100).reshape([10,10])
array2

    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, 36, 37, 38, 39],
           [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
           [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
           [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
           [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
           [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
           [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
array2.sum()

    4950
```

```
array2.sum(axis = 0)  # remember, axis 0 means columns.    (Most often you'd sum do

    array([450, 460, 470, 480, 490, 500, 510, 520, 530, 540])
```

```
array2.sum(axis = 1)  # axis 1 means rows

    array([ 45, 145, 245, 345, 445, 545, 645, 745, 845, 945])
```

```
array2.cumsum()

    array([   0,    1,    3,    6,   10,   15,   21,   28,   36,   45,   55,
             66,   78,   91,  105,  120,  136,  153,  171,  190,  210,  231,
            253,  276,  300,  325,  351,  378,  406,  435,  465,  496,  528,
            561,  595,  630,  666,  703,  741,  780,  820,  861,  903,  946,
            990, 1035, 1081, 1128, 1176, 1225, 1275, 1326, 1378, 1431, 1485,
           1540, 1596, 1653, 1711, 1770, 1830, 1891, 1953, 2016, 2080, 2145,
           2211, 2278, 2346, 2415, 2485, 2556, 2628, 2701, 2775, 2850, 2926,
           3003, 3081, 3160, 3240, 3321, 3403, 3486, 3570, 3655, 3741, 3828,
           3916, 4005, 4095, 4186, 4278, 4371, 4465, 4560, 4656, 4753, 4851,
           4950])
```

```
array2.min()
```

```
    0
```

```
array2.mean(axis = 0)
```

```
    array([45., 46., 47., 48., 49., 50., 51., 52., 53., 54.])
```

```
# You can also play with the code below to create arrays with specific values (remo

# Create an array of ones
#np.ones((3,4), dtype=np.int64)

# Create an array of zeros
#np.zeros((2,3,4),dtype=np.int16)

# Create an array with random values
#np.random.random((2,2))

# Create an empty array
#np.empty((3,2))

# Create a full array with a particular value
#np.full((2,2),9)

# Create an array of evenly-spaced values
#np.arange(10,25,5)

# Create an array of evenly-spaced values
#np.linspace(0,2,9)

# An array with values corresponding to an identity matrix of size 3
#np.eye(3)
#np.identity(3)
```

Slicing and indexing

Similar to other python data structures numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array

```
#Let's create a numpy array comprising of the integers 0 through 9
arr = np.arange(10)
arr
```

```
    array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# You can get the item at index 5
arr[5]
```

```
    5
```

```
# You can slice the array to get an array consisting of only those items lie betwee
arr[5:8]
```

```
    array([5, 6, 7])
```

```
# Let's look at another example:
arr2d=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d
```

```
    array([[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]])
```

```
# This returns a two dimensional array with shape (1,3)
arr2d[2:, :]
```

```
    array([[7, 8, 9]])
```

```
#By mixing integer indexes and slices, you get lower dimensional slices
# i.e. you get a one dimensional array with shape (3,)
arr2d[2, :]
```

```
    array([7, 8, 9])
```

```
array2 = np.arange(100).reshape([10,10])
```

```
array2[5:,5:]
```

```
    array([[55, 56, 57, 58, 59],
           [65, 66, 67, 68, 69],
           [75, 76, 77, 78, 79],
           [85, 86, 87, 88, 89],
           [95, 96, 97, 98, 99]])
```

```
array2[array2 > 50]
```

```
    array([51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
           68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
           85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

```
# index arrays based on conditions

array2 = np.arange(100).reshape([10,10])

array2[array2 < 30] = 0

array2
```

```
    array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
           [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
           [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
           [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
           [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
           [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
           [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
           [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
           [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
           [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

From you local computer drag and drop the winequlaity-red.csv into Colab storage

The we will use numpy to read the text data into the numpy array

```
redwinedata=np.genfromtxt('winequality-red.csv',delimiter=';',skip_header = 1)
redwinedata
```

```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-63-3b9b40130592> in <module>()
----> 1 redwinedata=np.genfromtxt('winequality-
red.csv',delimiter=';',skip_header = 1)
      2 redwinedata

2 frames
/usr/local/lib/python3.7/dist-packages/numpy/lib/_datasource.py in open(self,
path, mode, encoding, newline)
    531                                       encoding=encoding,
newline=newline)
    532         else:
--> 533             raise IOError("%s not found." % path)
    534
    535

OSError: winequality-red.csv not found.
```

Check the shape of the array

```
redwinedata.shape
```

```
#Add code
redwinedata.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-64-e0d71a37de06> in <module>()
      1 #Add code
----> 2 redwinedata.shape

NameError: name 'redwinedata' is not defined
```

SEARCH STACK OVERFLOW

Slice the redwindata array into two numpy array

with dataX array shape (1599,11) #11 columns from index 0 to 10

with dataY array shape (1599,1) #11 columns only index 11

```
#Add code
```

```
#Add code
```

7. MatPlotLib

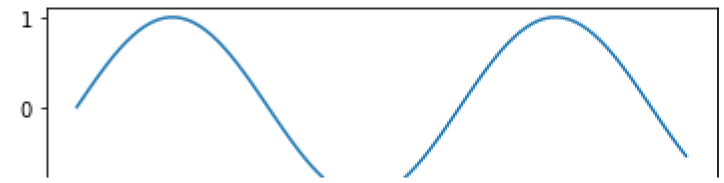
Plotting

MatPlotLib is a widely used plotting library. You can pass it data directly, either for data exploration purposes or to produce publication quality outputs. Also you should get to know it because Pandas calls it.

MATLAB-style Interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (`plt`) interface. For example, the following code will probably look quite familiar to MATLAB users:

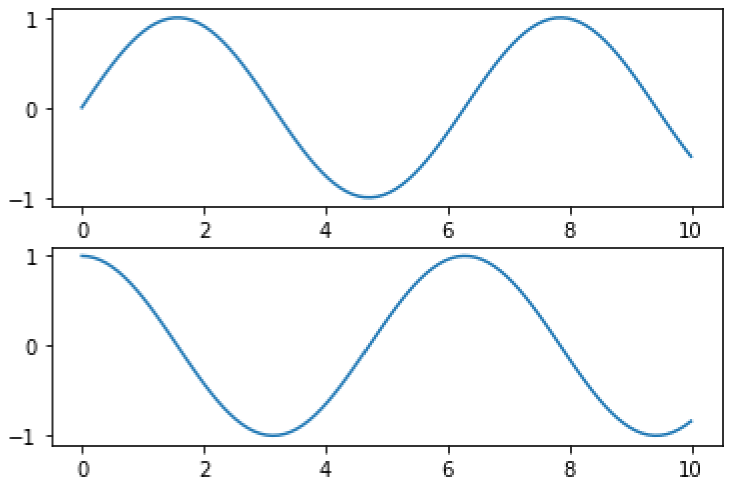
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)

plt.figure()  # create a plot figure

# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



Object-oriented interface

The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are *methods* of explicit `Figure` and `Axes` objects. To re-create the previous plot using this style of plotting, you might do the following:

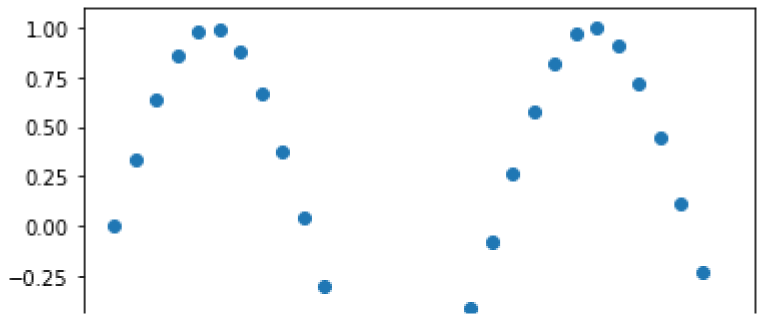
```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```



Scatter Plots with `plt.scatter`

A second, more powerful method of creating scatter plots is the `plt.scatter` function, which can be used very similarly to the `plt.plot` function:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.scatter(x, y, marker='o');
```



The primary difference of `plt.scatter` from `plt.plot` is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Let's show this by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the `alpha` keyword to adjust the transparency level:

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
print(x)
print(y)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis')
plt.colorbar();  # show color scale
```

```
 [ 1.76405235  0.40015721  0.97873798  2.2408932   1.86755799 -0.97727788
   0.95008842 -0.15135721 -0.10321885  0.4105985   0.14404357  1.45427351
   0.76103773  0.12167502  0.44386323  0.33367433  1.49407907 -0.20515826
   0.3130677  -0.85409574 -2.55298982  0.6536186   0.8644362  -0.74216502
   2.26975462 -1.45436567  0.04575852 -0.18718385  1.53277921  1.46935877
   0.15494743  0.37816252 -0.88778575 -1.98079647 -0.34791215  0.15634897
   1.23029068  1.20237985 -0.38732682 -0.30230275 -1.04855297 -1.42001794
  -1.70627019  1.9507754  -0.50965218 -0.4380743  -1.25279536  0.77749036
  -1.61389785 -0.21274028 -0.89546656  0.3869025  -0.51080514 -1.18063218
  -0.02818223  0.42833187  0.06651722  0.3024719  -0.63432209 -0.36274117
  -0.67246045 -0.35955316 -0.81314628 -1.7262826   0.17742614 -0.40178094
  -1.63019835  0.46278226 -0.90729836  0.0519454   0.72909056  0.12898291
   1.13940068 -1.23482582  0.40234164 -0.68481009 -0.87079715 -0.57884966
  -0.31155253  0.05616534 -1.16514984  0.90082649  0.46566244 -1.53624369
   1.48825219  1.89588918  1.17877957 -0.17992484 -1.07075262  1.05445173
  -0.40317695  1.22244507  0.20827498  0.97663904  0.3563664   0.70657317
   0.01050002  1.78587049  0.12691209  0.40198936]
 [ 1.8831507  -1.34775906 -1.270485    0.96939671 -1.17312341  1.94362119
  -0.41361898 -0.74745481  1.92294203  1.48051479  1.86755896  0.90604466
  -0.86122569  1.91006495 -0.26800337  0.8024564   0.94725197 -0.15501009
   0.61407937  0.92220667  0.37642553 -1.09940079  0.29823817  1.3263859
  -0.69456786 -0.14963454 -0.43515355  1.84926373  0.67229476  0.40746184
  -0.76991607  0.53924919 -0.67433266  0.03183056 -0.63584608  0.67643329
   0.57659082 -0.20829876  0.39600671 -1.09306151 -1.49125759  0.4393917
```

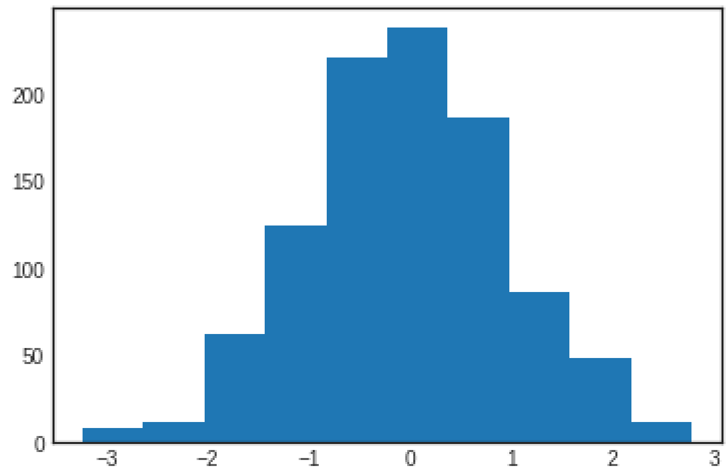
`plot` Versus `scatter`: A Note on Efficiency

Aside from the different features available in `plt.plot` and `plt.scatter`, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, `plt.plot` can be noticeably more efficient than `plt.scatter`. The reason is that `plt.scatter` has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In `plt.plot`, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, `plt.plot` should be preferred over `plt.scatter` for large datasets.

Histogram

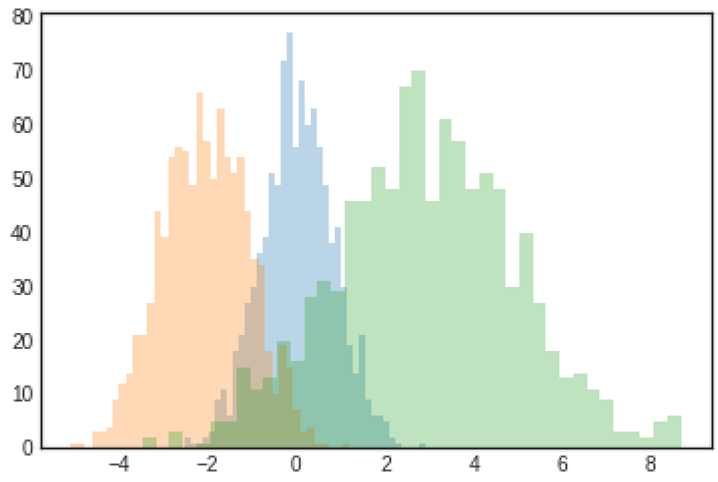
A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib's histogram function

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

data = np.random.randn(1000)


plt.hist(data);
```

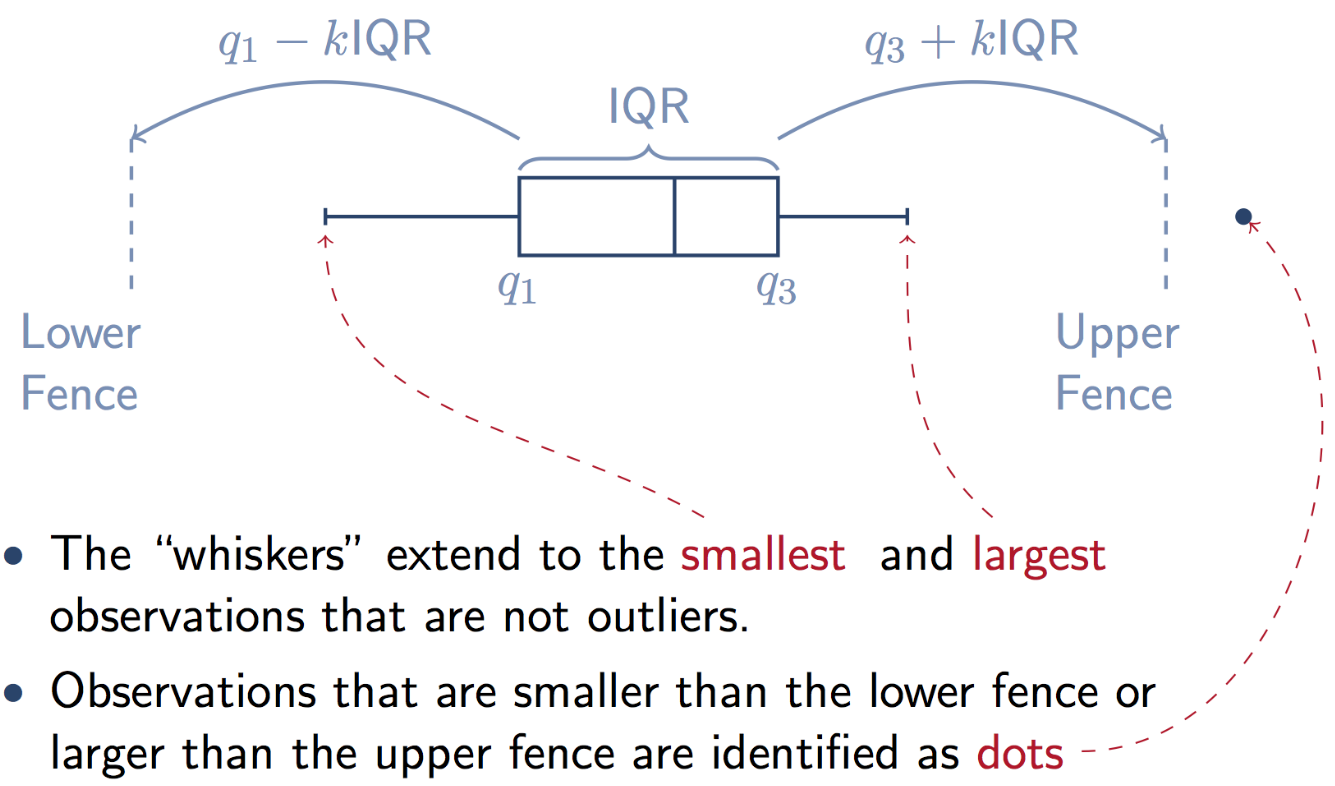


The `plt.hist` docstring has more information on other customization options available. I find this combination of `histtype='stepfilled'` along with some transparency `alpha` to be very useful when comparing histograms of several distributions:

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



Boxplot

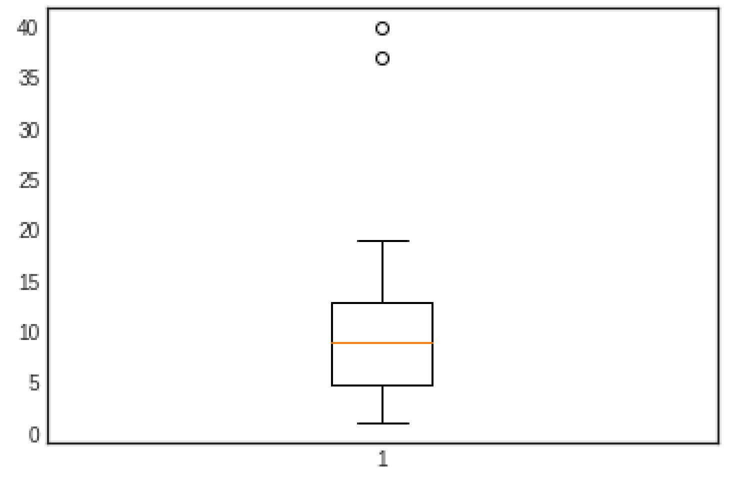


- The “whiskers” extend to the smallest and largest observations that are not outliers.
- Observations that are smaller than the lower fence or larger than the upper fence are identified as dots

```
# random integers between 1 to 20
arr = np.random.randint(1, 20, size=30)

# two outliers taken
arr1 = np.append(arr, [37, 40])

print('Thus the array becomes{}'.format(arr1))
```

```
Thus the array becomes[ 3  9  7 13 17  2  8  2  6 12 13  4  5 16 13 10 12 19 1
  3  3  1  2 13  9 37 40]
```

```
plt.boxplot(arr1)
fig = plt.figure(figsize =(10, 7))
plt.show()
```



```
<Figure size 720x504 with 0 Axes>
```

```
# finding the 1st quartile
q1 = np.quantile(arr1, 0.25)
print('q1',q1)
# finding the 3rd quartile
q3 = np.quantile(arr1, 0.75)
med = np.median(arr1)
print('med',med)
print('q3',q3)
# finding the iqr region
iqr = q3-q1
print('iqr',iqr)
# finding upper and lower whiskers
upper_bound = q3+(1.5*iqr)
lower_bound = q1-(1.5*iqr)
print(iqr, upper_bound, lower_bound)
```

```
q1 4.75
med 9.0
q3 13.0
iqr 8.25
8.25 25.375 -7.625
```

```
outliers = arr1[(arr1 <= lower_bound) | (arr1 >= upper_bound)]
print('The following are the outliers in the boxplot:{}'.format(outliers))
```

```
The following are the outliers in the boxplot:[37 40]
```

✓ 0s completed at 7:02 PM