Lesson_1_Intro_to_Python

August 28, 2020

1

[ ]:

2 Python-Lectures

2.1 Introduction

Python is a modern, robust, high level programming language. It is very easy to pick up even if you are completely new to programming.

Python, similar to other languages like matlab or R, is interpreted hence runs slowly compared to C++, Fortran or Java. However writing programs in Python is very quick. Python has a very large collection of libraries for everything from scientific computing to web services. It caters for object oriented and functional programming with module system that allows large and complex applications to be developed in Python.

These lectures are using jupyter notebooks which mix Python code with documentation. The python notebooks can be run on a webserver or stand-alone on a computer.

To give an indication of what Python code looks like, here is a simple bit of code that defines a set $N = \{1, 3, 4, 5, 7\}$ and calculates the sum of the squared elements of this set:

$$\sum_{i \in N} i^2 = 100$$

[ ]:
```
print("Hi")
```

```
Hi
```

[ ]:
```
N={1,3,4,5,7,8}
print('The sum of ?_i?N i*i =',sum( i**2 for i in N ) )
```

```
The sum of ?_i?N i*i = 164
```

[ ]:
```
x =2
print(x)
```

```
2
```

This is a tutorial style introduction to Python. For a quick reminder / summary of Python syntax the following Quick Reference Card may be useful. A longer and more detailed tutorial style introduction to python is available from the python site at: https://docs.python.org/3/tutorial/

2.1.1 Installing

Python runs on windows, linux, mac and other environments. There are many python distributions available. You can run this Notebook on Colab and not to worry about anything or run it on Jupyter Notebook and then you will need to install Anaconda and Docker. If you are installing Python, make sure to install Python 3.

3 Getting started

Python can be used like a calculator. Simply type in expressions to get them evaluated.

3.1 Basic syntax for statements

The basic rules for writing simple statments and expressions in Python are: * No spaces or tab characters allowed at the start of a statement: Indentation plays a special role in Python (see the section on control statements). For now simply ensure that all statements start at the beginning of the line. * The '#' character indicates that the rest of the line is a comment * Statements finish at the end of the line: * Except when there is an open bracket or paranthesis:

```
1+2
+3  #illegal continuation of the sum
(1+2
             + 3) # perfectly OK even with spaces
```

- A single backslash at the end of the line can also be used to indicate that a statement is still incomplete

```
1 + \
   2 + 3 # this is also OK
```

The jupyter notebook system for writting Python intersperses text (like this) with Python statements. Try typing something into the cell (box) below and press the 'run cell' button above (triangle+line symbol) to execute it.

[1]:
```
1+2+3
```

[1]:
```
6
```

Python has extensive help built in. You can execute **help()** for an overview or **help(x)** for any library, object or type **x** to get more information. For example:

[2]:
```
help(True)
```

```
Help on bool object:

class bool(int)
```

```
|  bool(x) -> bool
|
|  Returns True when the argument x is true, False otherwise.
|  The builtins True and False are the only two instances of the class bool.
|  The class bool is a subclass of the class int, and cannot be subclassed.
|
|  Method resolution order:
|      bool
|      int
|      object
|
|  Methods defined here:
|
|  __and__(self, value, /)
|      Return self&value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __str__(self, /)
|      Return str(self).
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  ----------------------------------------------------------------------
|  Methods inherited from int:
|
|  __abs__(self, /)
|      abs(self)
|
|  __add__(self, value, /)
|      Return self+value.
```

```
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __ceil__(…)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(…)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(…)
 |      default object formatter
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(…)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an
index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
```

```
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __lshift__(self, value, /)
|      Return self<<value.
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __round__(…)
|      Rounding an Integral returns itself.
```

```
 |      Rounding with an ndigits argument also returns an integer.
 |
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |
 |  __rrshift__(self, value, /)
 |      Return value>>self.
 |
 |  __rshift__(self, value, /)
 |      Return self>>value.
 |
 |  __rsub__(self, value, /)
 |      Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __sizeof__(…)
 |      Returns size in memory, in bytes
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(…)
 |      Truncating an Integral returns itself.
 |
 |  bit_length(…)
 |      int.bit_length() -> int
 |
 |      Number of bits necessary to represent self in binary.
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(…)
 |      Returns self, the complex conjugate of any int.
 |
 |  from_bytes(…) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |
 |      Return the integer represented by the given array of bytes.
 |
 |      The bytes argument must be a bytes-like object (e.g. bytes or
bytearray).
```

```
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
 |
 |  to_bytes(…)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |
 |      Return an array of bytes representing an integer.
 |
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative
integer
 |      is given, an OverflowError is raised.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from int:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

4 Variables & Values

A name that is used to denote something or a value is called a variable. In python, variables can be declared and values can be assigned to it as follows,

[3]:
```
x = 2          # anything after a '#' is a comment
y = 5
xy = 'Hey'
print(x+y, xy)
```

```
7 Hey
```

Multiple variables can be assigned with the same value.

[4]:
```
x = y = 1
print(x,y)
```

```
1 1
```

The basic types build into Python include `float` (floating point numbers), `int` (integers), `str` (unicode character strings) and `bool` (boolean). Some examples of each:

[11]:
```
2.0           # a simple floating point number
1e100         # a googol
-1234567890   # an integer
True or False # the two possible boolean values
'This is a string'
"It's another string"
print("""Triple quotes (also with '''), allow strings to break over multiple␣
 ↪lines.
Alternatively \n is a newline character (\t for tab, \\ is a single␣
 ↪backslash)""")
```

```
Triple quotes (also with '''), allow strings to break over multiple lines.
Alternatively
 is a newline character (        for tab, \ is a single backslash)
```

[12]:
```
# Cool integration of code and print with f (Python > 3.5)
x = 'sweet'
y = 10
print(f'One table spoon of {x} sugar has {y} grams')
```

```
One table spoon of sweet sugar has 10 grams
```

Python also has complex numbers that can be written as follows. Note that the brackets are required.

[6]:
```
complex(1,2)
(1+2j) # the same number as above
```

[6]:
```
(1+2j)
```

5 Operators

5.1 Arithmetic Operators

Symbol	Task Performed
+	Addition
-	Subtraction
/	division
%	mod (remainder after division)
	multiplication
//	floor division
	to the power of

[7]:
```
1+2
```

[7]:
```
3
```

[8]:
```
2-1
```

[8]:
```
1
```

[9]:
```
1*2
```

[9]:
```
2
```

[10]:
```
3/4
```

[10]:
```
0.75
```

Rounds down: (ie a // b= $\lfloor \frac{a}{b} \rfloor$)

[ ]:
```
3//4.0
```

[ ]:
```
0.0
```

[ ]:
```
15%10
```

[ ]:
```
5
```

Python natively allows (nearly) infinite length integers while floating point numbers are double precision numbers:

[ ]:
```
11**300
```

[ ]:
```
261701099618839990701703252897203834249164941695300026024080595582797205668538243449709034149678703258573888478674528670047399984728066419173100887481175131088859178611199467820892017514391176118142449566087795065414506696903625266973548309893688401647132648740379278764850687921263063710125924600570108432733800 1
```

[ ]:
```
11.0**300
```

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

OverflowError                                Traceback (most recent call␣
↪last)

<ipython-input-19-b61ab01789ad> in <module>
----> 1 11.0**300


OverflowError: (34, 'Result too large')
```

5.2 Relational Operators

Symbol	Task Performed
==	True, if it is equal
!=	True, if not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Note the difference between `==` (equality test) and `=` (assignment)

[ ]:
```
z = 2
z == 2
```

[ ]:
```
z > 2
```

[ ]:
```
z != 3
```

Comparisons can also be chained in the mathematically obvious way. The following will work as expected in Python:

[ ]:
```
x1 = z == 2
x2 = 0.5 < z <= 1
print(x1,x2)
```

5.3 Boolean and Bitwise Operators

|Operator|Meaning | | Symbol | Task Performed | |—-|— | |—-|—| |and| Logical and | | & | Bitwise And | |or | Logical or | | | | Bitwise OR | | | | | ^ | Exclusive or | |not | Not | | ~ | Negate | | | | | » | Right shift | | | | | « | Left shift |

[13]:
```
print( not (True and False), "==", not True or not False)
```

```
True == True
```

6 Built-in Functions

Python comes with a wide range of functions. However many of these are part of stanard libraries like the `math` library rather than built-in.

int( ) converts a number to an integer. This can be a single floating point number, integer or a string. For strings the base can optionally be specified:

[14]:
```
print(int(7.9999999), int('111',2),int('7') )
```

```
7 7 7
```

Similarly, the function **str( )** can be used to convert almost anything to a string

[15]:
```
print(str(True),str(1.2345678),str(-2))
```

```
True 1.2345678 -2
```

[16]:
```
x = str(2)
y = str(5)
x + y
```

[16]:
```
'25'
```

6.1 Mathematical functions

Mathematical functions include the usual suspects like logarithms, trigonometric fuctions, the constant π and so on.

[17]:
```
import math
math.sin(math.pi/4)
#from math import * # avoid having to put a math. in front of every␣
 ↪mathematical function
#sin(pi/2) # equivalent to the statement above
```

[17]:
```
0.7071067811865475
```

6.2 Simplifying Arithmetic Operations

round( ) function rounds the input value to a specified number of places or to the nearest integer.

[18]:
```
print( round(5.6231) )
print( round(4.55892, 2) )
```

```
6
4.56
```

divmod(x,y) outputs the quotient and the remainder in a tuple(you will be learning about it in the further chapters) in the format (quotient, remainder). Tupple is a generalized form of a vector, which can have elements of differnt types, for example (5,6,'Frodo')

[19]:
```
divmod(9,2)
```

[19]:
```
(4, 1)
```

6.3 Accepting User Inputs

input(prompt), prompts for and returns input as a string. A useful function to use in conjunction with this is **eval()** which takes a string and evaluates it as a python expression.

[22]:
```
abc =  input("abc = ")
abcValue=eval(abc)
print(abc,'=',abcValue)
```

```
abc = 3
3 = 3
```

7 Working with strings

7.1 The Print Statement

As seen previously, The **print()** function prints all of its arguments as strings, separated by spaces and follows by a linebreak:

- `print("Hello World")`
- `print("Hello",'World')`
- `print("Hello", <Variable Containing the String>)`

Note that **print** is different in old versions of Python (2.7) where it was a statement and did not need parenthesis around its arguments.

[23]:
```
x = 5
print(f"Precision equals to {x}")
```

```
Precision equals to 5
```

The print has some optional arguments to control where and how to print. This includes `sep` the separator (default space) and `end` (end charcter) and `file` to write to a file.

[24]:
```
print("Hello", "Happy", "Silly", "World",sep='...',end='!!')
```

```
Hello...Happy...Silly...World!!
```

7.2 String Formating

There are lots of methods for formating and manipulating strings built into python. Some of these are illustrated here.

String concatenation is the "addition" of two strings. Observe that while concatenating there will be no space between the strings.

[25]:
```
string1='World'
string2='!'
print('Hello' + ' ' + string1 + string2)
```

```
Hello World!
```

When referring to multiple variables parenthesis is used. Values are inserted in the order they appear in the paranthesis (more on tuples in the next lecture)

7.3 Other String Methods

Multiplying a string by an integer simply repeats it

[27]:
```
print("Hello World! "*5)
```

```
Hello World! Hello World! Hello World! Hello World! Hello World!
```

Strings can be tranformed by a variety of functions:

[28]:
```
s="hello wOrld"
print(s.capitalize())

print(s.upper())
print(s.lower())
print('|%s|' % "Hello World".center(30)) # center in 30 characters
print('|%s|'% "     lots  of space             ".strip()) # remove leading and␣
 ↪trailing whitespace
print("Hello World".replace("World","Class"))
```

```
Hello world
HELLO WORLD
hello world
|         Hello World          |
|lots  of space|
Hello Class
```

7.4 String comparison operations

Strings can be compared in lexicographical order with the usual comparisons. In addition the `in` operator checks for substrings:

[29]:
```
'abc' < 'bbc' <= 'bbc'
```

[29]: True

[30]:
```
"ABC" in "This is the ABC of Python"
```

[30]: True

7.5 Accessing parts of strings

Strings can be indexed with square brackets. Indexing starts from zero in Python.

[32]:
```
s = '123456789'
print('First charcter of',s,'is',s[0])
print('Last charcter of',s,'is',s[len(s)-1])
print('Last charcter of',s,'is',s[len(s)-1])
len(s)
print(s[9])
```

```
First charcter of 123456789 is 1
Last charcter of 123456789 is 9
Last charcter of 123456789 is 9


      ␣
↪---------------------------------------------------------------------------

       IndexError                                Traceback (most recent call␣
↪last)

       <ipython-input-32-ff18381f3055> in <module>()
         4 print('Last charcter of',s,'is',s[len(s)-1])
         5 len(s)
   ----> 6 print(s[9])


       IndexError: string index out of range
```

Negative indices can be used to start counting from the back

[33]:
```
print('First charcter of',s,'is',s[-len(s)])
print('Last charcter of',s,'is',s[-1])
```

```
First charcter of 123456789 is 1
Last charcter of 123456789 is 9
```

Finally a substring (range of characters) an be specified as using $a : b$ to specify the characters at index $a, a + 1, \ldots, b - 1$. Note that the last charcter is *not* included.

[34]:
```
print("First three charcters",s[0:3])
print("Next three characters",s[3:6])
```

```
First three charcters 123
Next three characters 456
```

An empty beginning and end of the range denotes the beginning/end of the string:

[35]:
```
print("First three characters", s[:3])
print("Last three characters", s[-3:])
```

```
First three characters 123
Last three characters 789
```

[ ]:
```
S = 'Taj Mahal is beautiful'
print([x for x in S if x.islower()]) # list of lower case charactes
words=S.split() # list of words
print("Words are:",words)
print("--".join(words)) # hyphenated
" ".join(w.capitalize() for w in words) # capitalise words
```

```
['a', 'j', 'a', 'h', 'a', 'l', 'i', 's', 'b', 'e', 'a', 'u', 't', 'i', 'f', 'u',
'l']
Words are: ['Taj', 'Mahal', 'is', 'beautiful']
Taj--Mahal--is--beautiful
```

[ ]:
```
'Taj Mahal Is Beautiful'
```

String Indexing and Slicing are similar to Lists which was explained in detail earlier.

[ ]:
```
print(S[4])
print(S[4:])
```

```
M
Mahal is beautiful
```

8 Data Structures

In simple terms, It is the the collection or group of data in a particular structure.

8.1 Lists

Lists are the most commonly used data structure. Think of it as a sequence of data that is enclosed in square brackets and data are separated by a comma. Each of these data can be accessed by calling it's index value.

Lists are declared by just equating a variable to '[ ]' or list. In lists we can change elements, in tupple we cannot.

[ ]:
```
a = []
```

[ ]:
```
type(a)
```

One can directly assign the sequence of data to a list x as shown.

[36]:
```
x = ['apple', 'orange']
```

8.1.1 Indexing

In python, indexing starts from 0 as already seen for strings. Thus now the list x, which has two elements will have apple at 0 index and orange at 1 index.

[37]:
```
x[0]
```

[37]:
```
'apple'
```

Indexing can also be done in reverse order. That is the last element can be accessed first. Here, indexing starts from -1. Thus index value -1 will be orange and index -2 will be apple.

[38]:
```
x[-1]
```

[38]:
```
'orange'
```

As you might have already guessed, x[0] = x[-2], x[1] = x[-1]. This concept can be extended towards lists with more many elements.

[39]:
```
y = ['carrot','potato']
```

Here we have declared two lists x and y each containing its own data. Now, these two lists can again be put into another list say z which will have it's data as two lists. This list inside a list is called as nested lists and is how an array would be declared which we will see later.

[40]:
```
z  = [x,y]
print( z )
```

```
[['apple', 'orange'], ['carrot', 'potato']]
```

Indexing in nested lists can be quite confusing if you do not understand how indexing works in python. So let us break it down and then arrive at a conclusion.

Let us access the data 'apple' in the above nested list. First, at index 0 there is a list ['apple','orange'] and at index 1 there is another list ['carrot','potato']. Hence z[0] should give us the first list which contains 'apple' and 'orange'. From this list we can take the second element (index 1) to get 'orange'

[41]:
```
print(z[0][1])
```

```
orange
```

Lists do not have to be homogenous. Each element can be of a different type:

[43]:
```
["this is a valid list",2,3.6,(1+2j),["a","sublist"]]
```

[43]:
```
['this is a valid list', 2, 3.6, (1+2j), ['a', 'sublist']]
```

8.1.2 Slicing

Indexing was only limited to accessing a single element, Slicing on the other hand is accessing a sequence of data inside the list. In other words "slicing" the list.

Slicing is done by defining the index values of the first element and the last element from the parent list that is required in the sliced list. It is written as parentlist[a : b] where a,b are the index values from the parent list. If a or b is not defined then the index value is considered to be the first value for a if a is not defined and the last value for b when b is not defined.

[44]:
```
num = [0,1,2,3,4,5,6,7,8,9]
print(num[0:4])
print(num[4:])
```

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

You can also slice a parent list with a fixed length or step length.

[ ]:
```
num[:9:3]
```

8.1.3 Built in List Functions

To find the length of the list or the number of elements in a list, **len( )** is used.

[46]:
```
len(num)
```

[46]:
```
10
```

If the list consists of all integer elements then **min( )** and **max( )** gives the minimum and maximum value in the list. Similarly **sum** is the sum

[47]:
```
print("min =",min(num),"  max =",max(num),"  total =",sum(num))
```

```
min = 0   max = 9   total = 45
```

[48]:
```
max(num)
```

[48]:
```
9
```

Lists can be concatenated by adding, '+' them. The resultant list will contain all the elements of the lists that were added. The resultant list will not be a nested list.

[49]:
```
[1,2,3] + [5,4,7]
```

[49]:
```
[1, 2, 3, 5, 4, 7]
```

There might arise a requirement where you might need to check if a particular element is there in a predefined list. Consider the below list.

[50]:
```
names = ['Earth','Air','Fire','Water']
```

To check if 'Fire' and 'Rajath' is present in the list names. A conventional approach would be to use a for loop and iterate over the list and use the if condition. But in python you can use 'a in b' concept which would return 'True' if a is present in b and 'False' if not.

[51]:
```
'Fire' in names
```

[51]: True

[52]:
```
'Space' in names
```

[52]: False

[53]:
```
nlist = ['1','94','93','1000']
print("max =",max(nlist))
print('min =',min(nlist))
```

```
max = 94
min = 1
```

Even if the numbers are declared in a string the first index of each element is considered and the maximum and minimum values are returned accordingly.

But if you want to find the **max( )** string element based on the length of the string then another parameter `key` can be used to specify the function to use for generating the value on which to sort. Hence finding the longest and shortest string in `mlist` can be doen using the `len` function:

[54]:
```
mlist = ['fire', 'globalization', 'key', 'price']
print('longest =',max(mlist, key=len))
print('shortest =',min(mlist, key=len))
```

```
longest = globalization
shortest = key
```

append( ) is used to add a single element at the end of the list.

[55]:
```
lst = [1,1,4,8,7]
lst.append(1)
print(lst)
```

```
[1, 1, 4, 8, 7, 1]
```

Appending a list to a list would create a sublist. If a nested list is not what is desired then the **extend( )** function can be used.

[56]:
```
lst.extend([10,11,12])
print(lst)
```

```
[1, 1, 4, 8, 7, 1, 10, 11, 12]
```

count( ) is used to count the number of a particular element that is present in the list.

[57]:
```
lst.count(1)
```

[57]:
```
3
```

index( ) is used to find the index value of a particular element. Note that if there are multiple elements of the same value then the first index value of that element is returned.

[58]:
```
lst.index(1)
```

[58]:
```
0
```

insert(x,y) is used to insert a element y at a specified index value x. **append( )** function made it only possible to insert at the end.

[59]:
```
lst.insert(5, 'name')
print(lst)
```

```
[1, 1, 4, 8, 7, 'name', 1, 10, 11, 12]
```

insert(x,y) inserts but does not replace element. If you want to replace the element with another element you simply assign the value to that particular index.

[60]:
```
lst[5] = 'Python'
print(lst)
```

```
[1, 1, 4, 8, 7, 'Python', 1, 10, 11, 12]
```

One can also remove element by specifying the element itself using the **remove( )** function.

[61]:
```
lst.remove('Python')
print(lst)
```

```
[1, 1, 4, 8, 7, 1, 10, 11, 12]
```

Alternative to **remove** function but with using index value is **del**

[62]:
```
del lst[1]
print(lst)
```

```
[1, 4, 8, 7, 1, 10, 11, 12]
```

[63]:
```
lst.sort()
print(lst)
print(sorted([3,2,1])) # another way to sort
```

```
[1, 1, 4, 7, 8, 10, 11, 12]
[1, 2, 3]
```

Python offers built in operation **sort( )** to arrange the elements in ascending order. Alternatively **sorted()** can be used to construct a copy of the list in sorted order

For descending order, By default the reverse condition will be False for reverse. Hence changing it to True would arrange the elements in descending order.

[64]:
```
lst.sort(reverse=True)
print(lst)
```

```
[12, 11, 10, 8, 7, 4, 1, 1]
```

Similarly for lists containing string elements, **sort( )** would sort the elements based on it's ASCII value in ascending and by specifying reverse=True in descending.

[65]:
```
names.sort()
print(names)
names.sort(reverse=True)
print(names)
```

```
['Air', 'Earth', 'Fire', 'Water']
['Water', 'Fire', 'Earth', 'Air']
```

To sort based on length key=len should be specified as shown.

[66]:
```
names.sort(key=len)
print(names)
print(sorted(names,key=len,reverse=True))
```

```
['Air', 'Fire', 'Water', 'Earth']
['Water', 'Earth', 'Fire', 'Air']
```

8.1.4 Copying a list

Assignment of a list does not imply copying. It simply creates a second reference to the same list. Most of new python programmers get caught out by this initially. Consider the following,

[67]:
```
a = [1, 2, 3]
b = a
b.append(5)
print(a, b)
```

```
[1, 2, 3, 5] [1, 2, 3, 5]
```

[68]:
```
a = [1, 2, 3]
b = a
a.append(5)
print(a, b)
```

```
[1, 2, 3, 5] [1, 2, 3, 5]
```

[69]:
```
import copy
a = [1, 2, 3]
b = copy.copy(a)
b.append(5)
print(a, b)
```

```
[1, 2, 3] [1, 2, 3, 5]
```

8.2 List comprehension

A very powerful concept in Python (that also applies to Tuples, sets and dictionaries as we will see below), is the ability to define lists using list comprehension (looping) expression. For example:

[70]:
```
[i**2 for i in [1,2,3]]
```

[70]:
```
[1, 4, 9]
```

As can be seen this constructs a new list by taking each element of the original `[1,2,3]` and squaring it. We can have multiple such implied loops to get for example:

[71]:
```
[10*i+j for i in [1,2,3] for j in [5,7]]
```

[71]:
```
[15, 17, 25, 27, 35, 37]
```

Finally the looping can be filtered using an **if** expression with the **for** - **in** construct.

[72]:
```
[10*i+j for i in [1,2,3] if i%2==1 for j in [4,5,7] if j >= i+4] # keep odd i␣
↪and  j larger than i+3 only
```

[72]:
```
[15, 17, 37]
```

NUMPY Numerical Arrays

[73]:
```
import numpy as np
```

[74]:
```
a = np.zeros((3,4))
a
```

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

[75]:
```
a.shape
```

[75]:
```
(3, 4)
```

[76]:
```
a.ndim  # equal to len(a.shape)
```

[76]:
```
2
```

[77]:
```
a.size
```

[77]:
```
12
```

Numpy Arrange

[81]:
```
np.arange(1, 5)
```

[81]:
```
array([1, 2, 3, 4])
```

[82]:
```
# Step parameter
```

[83]:
```
np.arange(1, 5, 0.5)
```

[83]:
```
array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

8.3 Strings

8.4 np.linspace

For this reason, it is generally preferable to use the `linspace` function instead of `arange` when working with floats. The `linspace` function returns an array containing a specific number of points evenly distributed between two values (note that the maximum value is *included*, contrary to `arange`):

[84]:
```
print(np.linspace(0, 5/3, 6))
```

```
[0.         0.33333333 0.66666667 1.         1.33333333 1.66666667]
```

8.5 np.rand and np.randn

A number of functions are available in NumPy's `random` module to create `ndarrays` initialized with random values. For example, here is a 3x4 matrix initialized with random floats between 0 and 1 (uniform distribution):

[86]:
```
np.random.rand(3,4)
```

[86]:
```
array([[0.428609  , 0.55602323, 0.22785091, 0.27713606],
       [0.98216109, 0.74550548, 0.23133551, 0.06978641],
       [0.22183181, 0.19424921, 0.3796727 , 0.94959511]])
```

Here's a 3x4 matrix containing random floats sampled from a univariate normal distribution (Gaussian distribution) of mean 0 and variance 1:

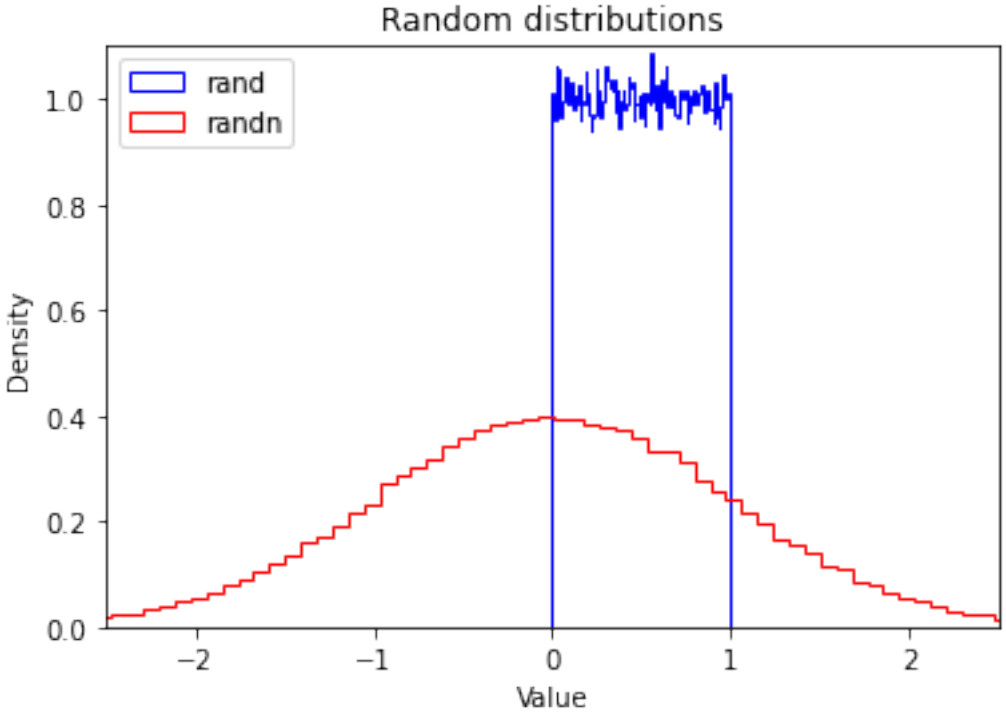
[87]:
```
np.random.randn(3,4)
```

[87]:
```
array([[-0.29255337, -1.98866821, -0.29782781,  1.57825447],
       [-0.05195013, -0.95946723, -0.76000631,  2.14104239],
       [-1.02628217,  0.93366514, -1.04606673,  1.55579743]])
```

To give you a feel of what these distributions look like, let's use matplotlib (see the matplotlib tutorial for more details):

[89]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

[90]:
```
plt.hist(np.random.rand(100000), density=True, bins=100, histtype="step",␣
 ↪color="blue", label="rand")
plt.hist(np.random.randn(100000), density=True, bins=100, histtype="step",␣
 ↪color="red", label="randn")
plt.axis([-2.5, 2.5, 0, 1.1])
plt.legend(loc = "upper left")
plt.title("Random distributions")
plt.xlabel("Value")
plt.ylabel("Density")
plt.show()
```



9 Reshaping an array

9.1 In place

Changing the shape of an `ndarray` is as simple as setting its `shape` attribute. However, the array's size must remain the same.

[92]:
```
g = np.arange(24)
print(g)
print("Rank:", g.ndim)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
Rank: 1
```

[93]:
```
g.shape = (6, 4)
print(g)
print("Rank:", g.ndim)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
Rank: 2
```

[94]:
```
g.shape = (2, 3, 4)
print(g)
print("Rank:", g.ndim)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
Rank: 3
```

9.2 reshape

The `reshape` function returns a new `ndarray` object pointing at the *same* data. This means that modifying one array will also modify the other.

[95]:
```
g2 = g.reshape(4,6)
print(g2)
print("Rank:", g2.ndim)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
Rank: 2
```

Set item at row 1, col 2 to 999 (more about indexing below).

[96]:
```
g2[1, 2] = 999
g2
```

[96]:
```
array([[  0,   1,   2,   3,   4,   5],
       [  6,   7, 999,   9,  10,  11],
       [ 12,  13,  14,  15,  16,  17],
       [ 18,  19,  20,  21,  22,  23]])
```

The corresponding element in `g` has been modified.

9.3 `ravel`

Finally, the `ravel` function returns a new one-dimensional `ndarray` that also points to the same data:

[97]:
```
g.ravel()
```

[97]:
```
array([  0,   1,   2,   3,   4,   5,   6,   7, 999,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23])
```

10 Stacking arrays

It is often useful to stack together different arrays. NumPy offers several functions to do just that. Let's start by creating a few arrays.

[98]:
```
q1 = np.full((3,4), 1.0)
q1
```

[98]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

[99]:
```
q2 = np.full((4,4), 2.0)
q2
```

[99]:
```
array([[2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.],
       [2., 2., 2., 2.]])
```

[100]:
```
q3 = np.full((3,4), 3.0)
q3
```

[100]:
```
array([[3., 3., 3., 3.],
       [3., 3., 3., 3.],
       [3., 3., 3., 3.]])
```

10.1 `vstack`

Now let's stack them vertically using `vstack`:

[101]:
```
q4 = np.vstack((q1, q2, q3))
q4
```

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

[102]:
```
q4.shape
```

[102]:
```
(10, 4)
```

This was possible because q1, q2 and q3 all have the same shape (except for the vertical axis, but that's ok since we are stacking on that axis).

10.2 `hstack`

We can also stack arrays horizontally using `hstack`:

[103]:
```
q5 = np.hstack((q1, q3))
q5
```

[103]:
```
array([[1., 1., 1., 1., 3., 3., 3., 3.],
       [1., 1., 1., 1., 3., 3., 3., 3.],
       [1., 1., 1., 1., 3., 3., 3., 3.]])
```

[ ]:
```
q5.shape
```

[ ]:
```
(3, 8)
```

11 Arithmetic operations

All the usual arithmetic operators (`+`, `-`, `*`, `/`, `//`, `**`, etc.) can be used with `ndarrays`. They apply *elementwise*:

[104]:
```
a = np.array([14, 23, 32, 41])
b = np.array([5,  4,  3,  2])
print("a + b  =", a + b)
```

```
print("a - b  =", a - b)
print("a * b  =", a * b)
print("a / b  =", a / b)
print("a // b  =", a // b)
print("a % b  =", a % b)
print("a ** b =", a ** b)
```

```
a + b  = [19 27 35 43]
a - b  = [ 9 19 29 39]
a * b  = [70 92 96 82]
a / b  = [ 2.8         5.75       10.66666667 20.5       ]
a // b  = [ 2  5 10 20]
a % b  = [4 3 2 1]
a ** b = [537824 279841  32768   1681]
```

Note that the multiplication is *not* a matrix multiplication. We will discuss matrix operations below.

The arrays must have the same shape. If they do not, NumPy will apply the *broadcasting rules*.

And also, very simply:

12 Mathematical and statistical functions

Many mathematical and statistical functions are available for **ndarrays**.

12.1 ndarray methods

Some functions are simply **ndarray** methods, for example:

[106]:
```
a = np.array([[-2.5, 3.1, 7], [10, 11, 12]])
print(a)
print("mean =", a.mean())
```

```
[[-2.5  3.1  7. ]
 [10.  11.  12. ]]
mean = 6.766666666666667
```

Note that this computes the mean of all elements in the **ndarray**, regardless of its shape.

Here are a few more useful **ndarray** methods:

[107]:
```
for func in (a.min, a.max, a.sum, a.prod, a.std, a.var):
    print(func.__name__, "=", func())
```

```
min = -2.5
max = 12.0
sum = 40.6
prod = -71610.0
std = 5.084835843520964
var = 25.855555555555554
```

These functions accept an optional argument **axis** which lets you ask for the operation to be performed on elements along the given axis. For example:

[108]:
```
c=np.arange(24).reshape(2,3,4)
c
```

[108]:
```
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]]])
```

[109]:
```
c.sum(axis=0)  # sum across matrices
```

[109]:
```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

[110]:
```
c.sum(axis=1)  # sum across rows
```

[110]:
```
array([[12, 15, 18, 21],
       [48, 51, 54, 57]])
```

You can also sum over multiple axes:

[111]:
```
c.sum(axis=(0,2))  # sum across matrices and columns
```

[111]:
```
array([ 60,  92, 124])
```

[112]:
```
0+1+2+3 + 12+13+14+15, 4+5+6+7 + 16+17+18+19, 8+9+10+11 + 20+21+22+23
```

[112]:
```
(60, 92, 124)
```

12.2 Universal functions

NumPy also provides fast elementwise functions called *universal functions*, or **ufunc**. They are vectorized wrappers of simple functions. For example **square** returns a new **ndarray** which is a copy of the original **ndarray** except that each element is squared:

[ ]:
```
a = np.array([[-2.5, 3.1, 7], [10, 11, 12]])
np.square(a)
```

[ ]:
```
array([[  6.25,   9.61,  49.  ],
       [100.  , 121.  , 144.  ]])
```

Here are a few more useful unary ufuncs:

[113]:
```
print("Original ndarray")
print(a)
for func in (np.abs, np.sqrt, np.exp, np.log, np.sign, np.ceil, np.modf, np.
 ↪isnan, np.cos):
    print("\n", func.__name__)
    print(func(a))
```

```
Original ndarray
[[-2.5  3.1  7. ]
 [10.  11.  12. ]]

 absolute
[[ 2.5  3.1  7. ]
 [10.  11.  12. ]]

 sqrt
[[       nan 1.76068169 2.64575131]
 [3.16227766 3.31662479 3.46410162]]

 exp
[[8.20849986e-02 2.21979513e+01 1.09663316e+03]
 [2.20264658e+04 5.98741417e+04 1.62754791e+05]]

 log
[[       nan 1.13140211 1.94591015]
 [2.30258509 2.39789527 2.48490665]]

 sign
[[-1.  1.  1.]
 [ 1.  1.  1.]]

 ceil
[[-2.  4.  7.]
 [10. 11. 12.]]

 modf
(array([[-0.5,  0.1,  0. ],
       [ 0. ,  0. ,  0. ]]), array([[-2.,  3.,  7.],
       [10., 11., 12.]]))

 isnan
[[False False False]
 [False False False]]

 cos
[[-0.80114362 -0.99913515  0.75390225]
 [-0.83907153  0.0044257   0.84385396]]
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:5: RuntimeWarning:
invalid value encountered in sqrt
  """
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:5: RuntimeWarning:
invalid value encountered in log
  """
```

13 Saving and loading

NumPy makes it easy to save and load `ndarrays` in text format.

[117]:
```
a_loaded = np.load("my_array.npy")
a_loaded
```

[117]:
```
array([[0.37789264, 0.29762478, 0.76082531],
       [0.98751579, 0.60394282, 0.2464924 ]])
```

13.1 Text format

Let's try saving the array in text format:

[118]:
```
np.savetxt("my_array.csv", a)
```

Now let's look at the file content:

[119]:
```
with open("my_array.csv", "rt") as f:
    print(f.read())
```

```
3.778926418185427627e-01 2.976247777923574089e-01 7.608253110541550734e-01
9.875157948610674419e-01 6.039428199329420766e-01 2.464923961988854106e-01
```

This is a CSV file with tabs as delimiters. You can set a different delimiter:

[120]:
```
np.savetxt("my_array.csv", a, delimiter=",")
```

To load this file, just use `loadtxt`:

[121]:
```
a_loaded = np.loadtxt("my_array.csv", delimiter=",")
a_loaded
```

[121]:
```
array([[0.37789264, 0.29762478, 0.76082531],
       [0.98751579, 0.60394282, 0.2464924 ]])
```

14 Control Flow Statements

The key thing to note about Python's control flow statements and program structure is that it uses *indentation* to mark blocks. Hence the amount of white space (space or tab characters) at the

start of a line is very important. This generally helps to make code more readable but can catch out new users of python.

14.1 Conditionals

14.1.1 If

```
python if some_condition:     code block
```

[122]:
```
x = 12
if x < 10:
    print("Hello")

print("no big deal")
```

```
no big deal
```

14.1.2 If-else

```
python if some_condition:     algorithm else:     algorithm
```

[123]:
```
x = 12
if 10 < x < 11:
    print("hello")
else:
    print("world")
```

```
world
```

14.1.3 Else if

```
python if some_condition:       algorithm elif some_condition:     algorithm
else:     algorithm
```

[124]:
```
x = 10
y = 12
if x > y:
    print("x>y")
elif x < y:
    print("x<y")
else:
    print("x=y")
```

```
x<y
```

if statement inside a if statement or if-elif or if-else are called as nested if statements.

[125]:
```
x = 10
y = 12
if x > y:
    print( "x>y")
```

```
# Next we test for this because
elif x < y:
    print( "x<y")
    if x==10:
        print ("x=10")
    else:
        print ("invalid")
else:
    print ("x=y")
```

```
x<y
x=10
```

14.2 Loops

14.2.1 For

`python for variable in something:  algorithm`

When looping over integers the **range()** function is useful which generates a range of integers: * range(n) = 0, 1, ..., n-1 * range(m,n)= m, m+1, ..., n-1 * range(m,n,s)= m, m+s, m+2s, ..., m + ((n-m-1)//s) * s

[ ]:
```
for ch in 'abc':
    print(ch)
total = 0
for i in range(5):
    total += i
print("total =",total)
for i,j in [(1,2),(3,1)]:
    # 10 + 1^2 + 3^2 = 14
    print(i,j)
    print(total)
    total += i**j
print("total =",total)
```

```
a
b
c
total = 10
1 2
10
3 1
11
total = 14
```

In the above example, i iterates over the 0,1,2,3,4. Every time it takes each value and executes the algorithm inside the loop. It is also possible to iterate over a nested list illustrated below.

[ ]:
```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list1 in list_of_lists:
        print(list1)
```

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

A use case of a nested for loop in this case would be,

[ ]:
```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
total=0
for list1 in list_of_lists:
    for x in list1:
        print(x)
        total = total+x
        print(total)
print(total)
```

```
1
1
2
3
3
6
4
10
5
15
6
21
7
28
8
36
9
45
45
```

14.2.2 While

`python while some_condition:  algorithm`

[ ]:
```
i = 1
while i < 3:
    print(i ** 2)
    i = i+1
print('Bye')
```

```
1
4
Bye
```

14.2.3 Break

As the name says. It is used to break out of a loop when a condition becomes true when executing the loop.

[ ]:
```
for i in range(100):
    print(i)
    if i>=7:
        break
```

```
0
1
2
3
4
5
6
7
```

15 Functions

Functions can represent mathematical functions. More importantly, in programmming functions are a mechansim to allow code to be re-used so that complex programs can be built up out of simpler parts.

This is the basic syntax of a function

```
python def funcname(arg1, arg2,... argN):     ''' Document String'''
statements     return <value>
```

Read the above syntax as, A function by name "funcname" is defined, which accepts arguements "arg1,arg2,....argN". The function is documented and it is ''Document String" '. The function after executing the statements returns a "value".

Return values are optional (by default every function returns **None** if no return statement is executed)

[ ]:
```
print("Hello Jack.")
print("Jack, how are you?")
```

```
Hello Jack.
Jack, how are you?
```

Instead of writing the above two statements every single time it can be replaced by defining a function which would do the job in just one line.

Defining a function firstfunc().

[ ]:
```
def firstfunc():
    print("Hello Jack.")
    print("Jack, how are you?")
firstfunc() # execute the function
```

```
Hello Jack.
Jack, how are you?
```

firstfunc() every time just prints the message to a single person. We can make our function **firstfunc()** to accept arguements which will store the name and then prints respective to that accepted name. To do so, add a argument within the function as shown.

[ ]:
```
firstfunc()
```

```
Hello Jack.
Jack, how are you?
```

[ ]:
```
def firstfunc(username):
    print("Hello %s." % username)
    print(username + ',' ,"how are you?")
```

[ ]:
```
name1 = 'sally' # or use input('Please enter your name : ')
```

So we pass this variable to the function **firstfunc()** as the variable username because that is the variable that is defined for this function. i.e name1 is passed as username.

[ ]:
```
firstfunc(name1)
```

```
Hello sally.
sally, how are you?
```

15.1 Return Statement

When the function results in some value and that value has to be stored in a variable or needs to be sent back or returned for further operation to the main algorithm, a return statement is used.

[ ]:
```
def times(x,y):
    z = x*y
    return z
```

The above defined **times( )** function accepts two arguements and return the variable z which contains the result of the product of the two arguements

[ ]:
```
c = times(4,5)
print(c)
```

```
20
```

The z value is stored in variable c and can be used for further operations.

Instead of declaring another variable the entire statement itself can be used in the return statement as shown.

[ ]:
```
def times(x,y):
    '''This multiplies the two input arguments'''
    return x*y
```

[ ]:
```
c = times(4,5)
print(c)
x = c + 5
print(x)
```

```
20
25
```

Multiple variable can also be returned. However this tends not to be very readable when returning many value, and can easily introduce errors when the order of return values is interpreted incorrectly.

[ ]:
```
eglist = [10,50,30,12,6,8,100]
```

[ ]:
```
def egfunc(eglist):
    highest = max(eglist)
    lowest = min(eglist)
    first = eglist[0]
    last = eglist[-1]
    return highest,lowest,first,last
```

If the function is just called without any variable for it to be assigned to, the result is returned inside a tuple. But if the variables are mentioned then the result is assigned to the variable in a particular order which is declared in the return statement.

[ ]:
```
egfunc(eglist)
```

[ ]:
```
(100, 6, 10, 100)
```

[ ]:
```
egfunc(eglist)[0]
```

[ ]:
```
100
```

15.2 Default arguments

When an argument of a function is common in majority of the cases this can be specified with a default value. This is also called an implicit argument.

[ ]:
```
def implicitadd(x,y=3,z=0):
    print("%d + %d + %d = %d"%(x,y,z,x+y+z))
    return x+y+z
```

implicitadd( ) is a function accepts up to three arguments but most of the times the first argument needs to be added just by 3. Hence the second argument is assigned the value 3 and the third argument is zero. Here the last two arguments are default arguments.

Now if the second argument is not defined when calling the **implicitadd( )** function then it considered as 3.

[ ]:
```
implicitadd(4)
```

```
4 + 3 + 0 = 7
```

[ ]: 7

However we can call the same function with two or three arguments. A useful feature is to explicitly name the argument values being passed into the function. This gives great flexibility in how to call a function with optional arguments. All off the following are valid:

[ ]:
```
implicitadd(4,4)
implicitadd(4,5,6)
implicitadd(4,z=7)
implicitadd(2,y=1,z=9)
implicitadd(x=1)
```

```
4 + 4 + 0 = 8
4 + 5 + 6 = 15
4 + 3 + 7 = 14
2 + 1 + 9 = 12
1 + 3 + 0 = 4
```

[ ]: 4

15.3 Global and Local Variables

Whatever variable is declared inside a function is local variable and outside the function in global variable.

[ ]:
```
eg1 = [1,2,3,4,5]
```

In the below function we are appending a element to the declared list inside the function. eg2 variable declared inside the function is a local variable.

[ ]:
```
def egfunc1():
    x=1
    def thirdfunc():
        x=2
        print("Inside thirdfunc x =", x)
    thirdfunc()
    print("Outside x =", x)
```

[ ]:
```
egfunc1()
```

```
Inside thirdfunc x = 2
Outside x = 1
```

If a **global** variable is defined as shown in the example below then that variable can be called from anywhere. Global values should be used sparingly as they make functions harder to re-use.

[ ]:
```
eg3 = [1,2,3,4,5]
```

[ ]:
```
def egfunc1():
    x = 1.0 # local variable for egfunc1
    def thirdfunc():
        global x # globally defined variable
        x = 2.0
        print("Inside thirdfunc x =", x)
    thirdfunc()
    print("Outside x =", x)
```

[ ]:
```
egfunc1()
print("Globally defined x =",x)
```

```
Inside thirdfunc x = 2.0
Outside x = 1.0
Globally defined x = 2.0
```

15.4 Lambda Functions

These are small functions which are not defined with any name and carry a single expression whose result is returned. Lambda functions comes very handy when operating with lists. These function are defined by the keyword **lambda** followed by the variables, a colon and the respective expression.

[ ]:
```
z = lambda x: x * x
```

[ ]:
```
z(8)
```

[ ]:
```
64
```

15.4.1 Composing functions

Lambda functions can also be used to compose functions

[ ]:
```
def double(x):
    return 2*x
def square(x):
    return x*x
def f_of_g(f,g):
    "Compose two functions of a single variable"
    return lambda x: f(g(x))
doublesquare= f_of_g(double,square)
print("doublesquare is a",type(doublesquare))
```

```
doublesquare(3)
# 2*(3^2) = 18
```

```
doublesquare is a <class 'function'>
```

[ ]:
```
18
```

15.4.2 Plotting

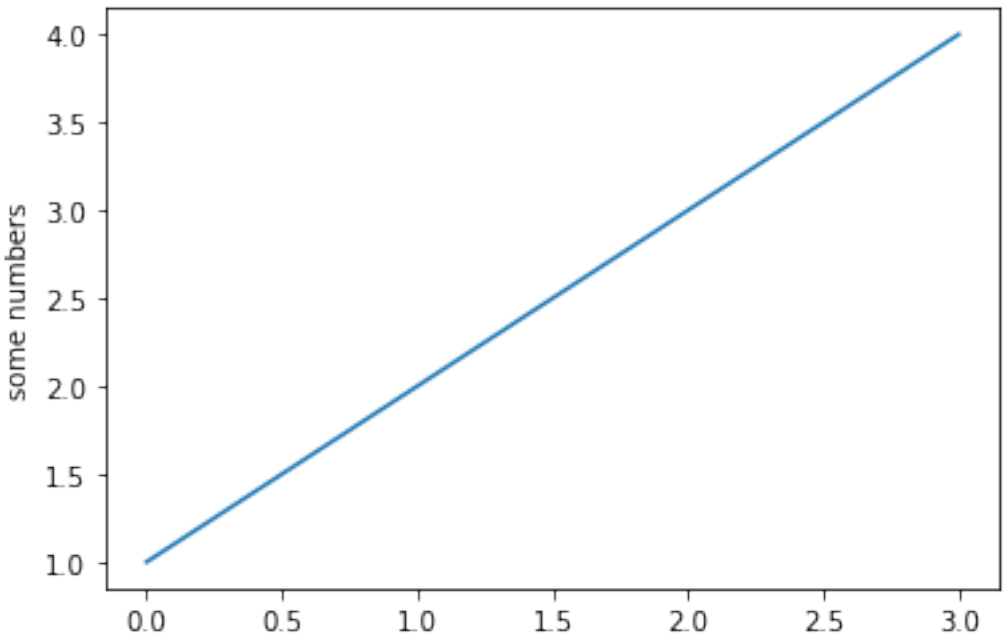
There are lots of configuration options for the **matplotlib** library that we are using here. For more information see [http://matplotlib.org/users/beginner.html]

To get started we need the following bit of 'magic' to make the plotting work:

[ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

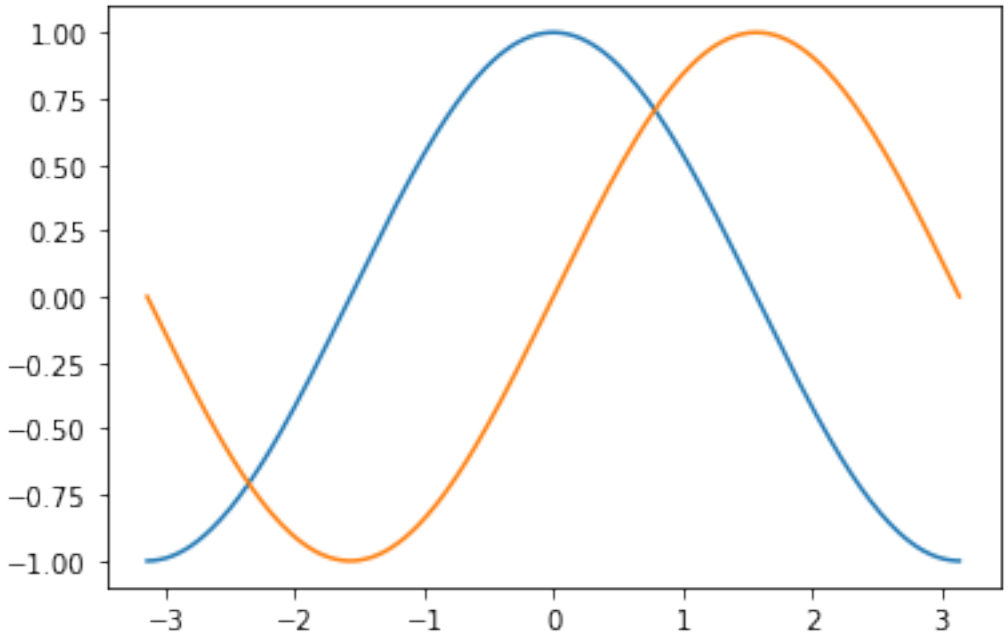
Now we can try something simple:

[ ]:
```
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



[ ]:
```
# A slightly more complicated plot with the help of numpy
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)
```

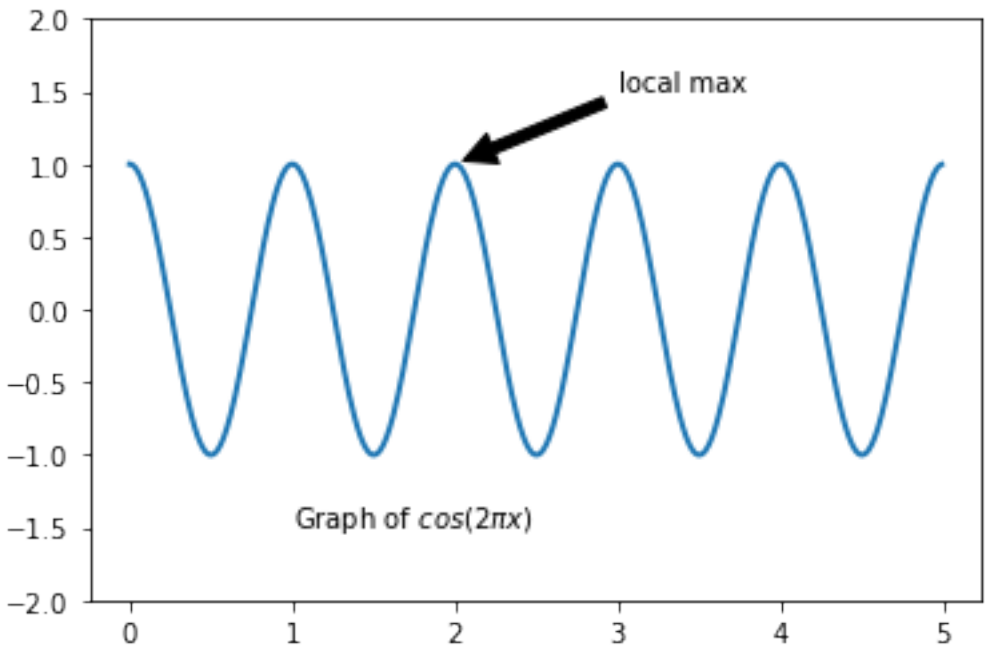
```
plt.plot(X, C)
plt.plot(X, S)

plt.show()
```



[ ]:

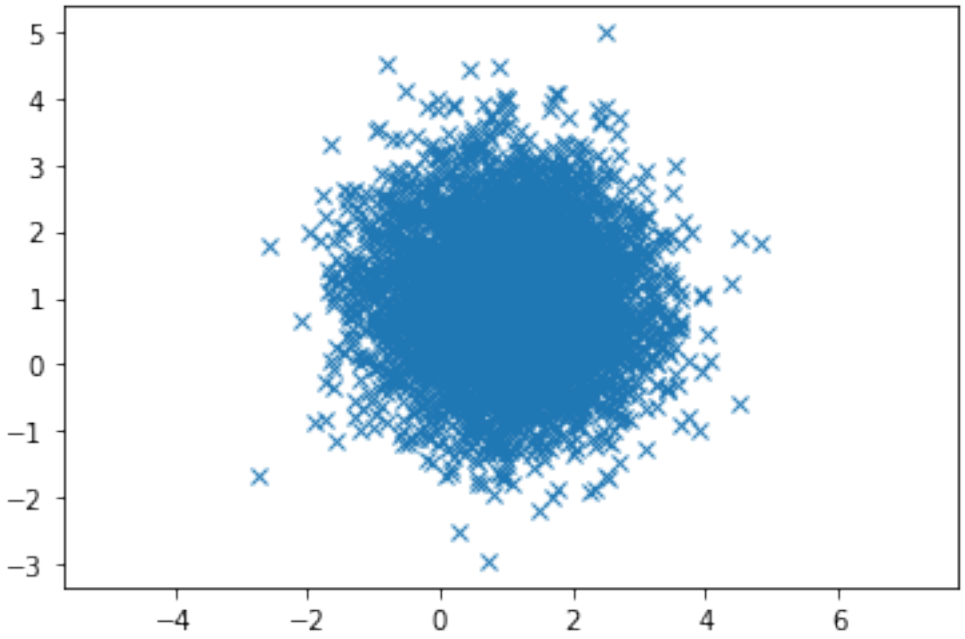
Annotating plots can be done with methods like **text()** to place a label and **annotate()**. For example:

[ ]:
```
t = np.arange(0.0, 5.0, 0.01)
line, = plt.plot(t, np.cos(2*np.pi*t), lw=2)
plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05),
             )
# text can include basic LaTeX commands - but need to mark
# string as raw (r"") or escape '\' (by using '\\')
plt.text(1,-1.5,r"Graph of $cos(2\pi x)$")
plt.ylim(-2,2)
plt.show()
```



Here is an example of how to create a basic surface contour plot.

[ ]:
```
import matplotlib.pyplot as plt
mean = [1,1]
cov= [[1,0],[0,1]]
x, y = np.random.multivariate_normal(mean, cov, 5000).T
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```



[ ]:
```
import matplotlib.pyplot as plt
mean = [1,1]
cov= [[0,1],[0,1]]
```

[ ]:
```
x = np.arange(-3.0, 3.0, delta)
y = np.arange(-2.0, 2.0, delta)
X, Y = np.meshgrid(x, y) # define mesh of points
x = np.random.multivariate_normal(mean, cov, (3, 3))
```

Load and Save files

[ ]:
```
import csv
import pandas as pd
import numpy as np
price = pd.read_csv('C:/Users/ilyar/Desktop/ML_JHU/Finished/Consumer_Complaints.
↪csv')
```

[ ]:
```
price
```

[ ]:
```

```

[ ]:
```
#Randomly draw 1% of the sample
oneperc = price.sample(frac=0.01, replace=False)
```

[ ]:
```
# Get first 5 observations in a separate dataset
```

[ ]:
```
price5 = price.iloc[0:5]
price5
```

16 Setup

First, let's import `pandas`. People usually import it as `pd`:

[ ]:
```
import pandas as pd
```

17 `Series` objects

The `pandas` library contains these useful data structures: * `Series` objects, that we will discuss now. A `Series` object is 1D array, similar to a column in a spreadsheet (with a column name and row labels). * `DataFrame` objects. This is a 2D table, similar to a spreadsheet (with column names and row labels). * `Panel` objects. You can see a `Panel` as a dictionary of `DataFrames`. These are less used, so we will not discuss them here.

17.1 Creating a `Series`

Let's start by creating our first `Series` object!

[ ]:
```
s = pd.Series([2,-1,3,5])
s
```

[ ]:
```
0    2
1   -1
2    3
3    5
dtype: int64
```

17.2 Similar to a 1D `ndarray`

`Series` objects behave much like one-dimensional NumPy `ndarrays`, and you can often pass them as parameters to NumPy functions:

[ ]:
```
import numpy as np
np.exp(s)
```

[ ]:
```
0      7.389056
1      0.367879
2     20.085537
3    148.413159
dtype: float64
```

Arithmetic operations on `Series` are also possible, and they apply *elementwise*, just like for `ndarrays`:

[ ]:
```
s + [1000,2000,3000,4000]
```

[ ]:
```
0    1002
1    1999
2    3003
3    4005
dtype: int64
```

Similar to NumPy, if you add a single number to a `Series`, that number is added to all items in the `Series`. This is called * broadcasting*:

[ ]:
```
s + 1000
```

[ ]:
```
0    1002
1     999
2    1003
3    1005
dtype: int64
```

The same is true for all binary operations such as `*` or `/`, and even conditional operations:

[ ]:
```
s < 0
```

[ ]:
```
0    False
1     True
2    False
3    False
dtype: bool
```

17.3 Index labels

Each item in a `Series` object has a unique identifier called the *index label*. By default, it is simply the rank of the item in the `Series` (starting at `0`) but you can also set the index labels manually:

[ ]:
```
s2 = pd.Series([68, 83, 112, 68], index=["alice", "bob", "charles", "darwin"])
s2
```

[ ]:
```
alice       68
bob         83
charles    112
darwin      68
dtype: int64
```

You can then use the `Series` just like a `dict`:

[ ]:
```
s2["bob"]
```

[ ]:
```
83
```

You can still access the items by integer location, like in a regular array:

[ ]:
```
s2[1]
```

[ ]:
```
83
```

To make it clear when you are accessing by label or by integer location, it is recommended to always use the `loc` attribute when accessing by label, and the `iloc` attribute when accessing by integer location:

[ ]:
```
s2.loc["bob"]
```

[ ]:
```
83
```

[ ]:
```
s2.iloc[1]
```

[ ]:
```
83
```

Slicing a `Series` also slices the index labels:

[ ]:
```
s2.iloc[1:3]
```

[ ]:
```
bob         83
charles    112
dtype: int64
```

This can lead to unexpected results when using the default numeric labels, so be careful:

[ ]:
```
surprise = pd.Series([1000, 1001, 1002, 1003])
surprise
```

[ ]:
```
0    1000
1    1001
2    1002
3    1003
dtype: int64
```

[ ]:
```
surprise_slice = surprise[2:]
surprise_slice
```

[ ]:
```
2    1002
3    1003
dtype: int64
```

Oh look! The first element has index label `2`. The element with index label `0` is absent from the slice:

[ ]:
```
try:
    surprise_slice[0]
```

```
except KeyError as e:
    print("Key error:", e)
```

```
Key error: 0
```

But remember that you can access elements by integer location using the `iloc` attribute. This illustrates another reason why it's always better to use `loc` and `iloc` to access `Series` objects:

[ ]:
```
surprise_slice.iloc[0]
```

[ ]:
```
1002
```

[ ]:
```

```

Tools - matplotlib

This notebook demonstrates how to use the matplotlib library to plot beautiful graphs.

18 Plotting your first graph

First we need to import the `matplotlib` library.

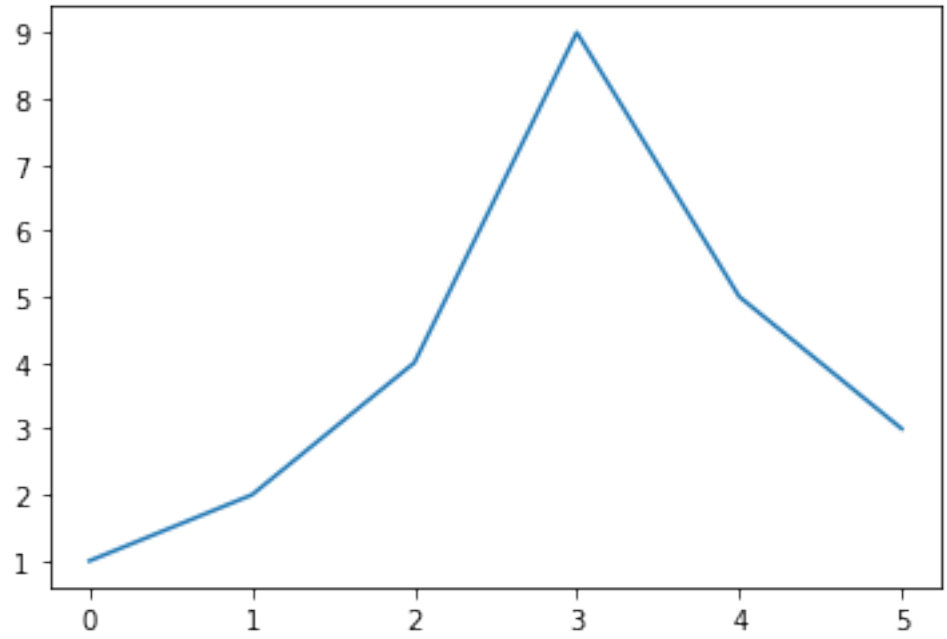
[ ]:
```
import matplotlib
```

Matplotlib can output graphs using various backend graphics libraries, such as Tk, wxPython, etc. When running python using the command line, the graphs are typically shown in a separate window. In a Jupyter notebook, we can simply output the graphs within the notebook itself by running the `%matplotlib inline` magic command.

[ ]:
```
%matplotlib inline
# matplotlib.use("TKAgg")  # use this instead in your program if you want to␣
↪use Tk as your graphics backend.
```

Now let's plot our first graph! :)

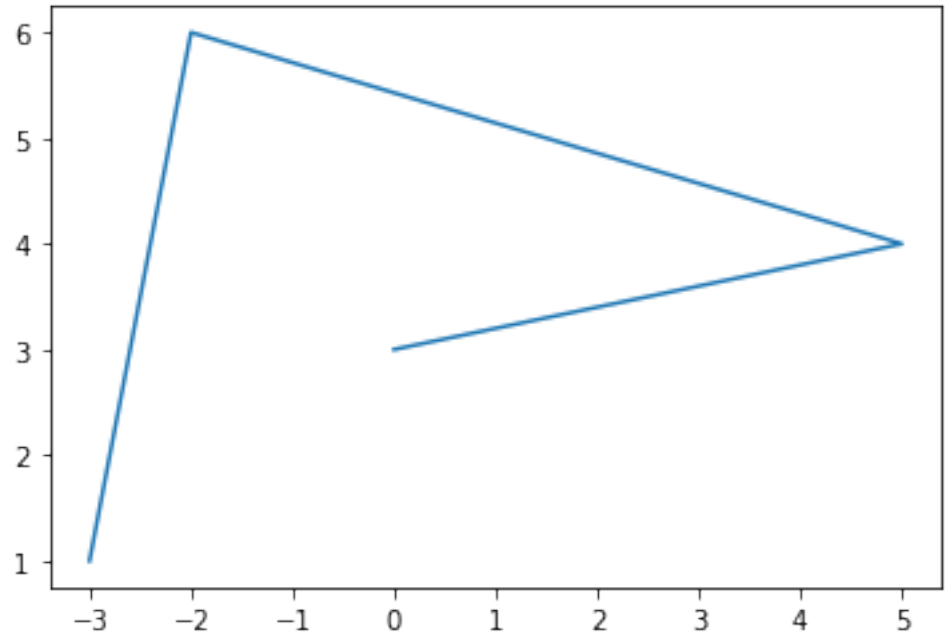
[ ]:
```
import matplotlib.pyplot as plt
plt.plot([1, 2, 4, 9, 5, 3])
plt.show()
```



Yep, it's as simple as calling the `plot` function with some data, and then calling the `show` function!

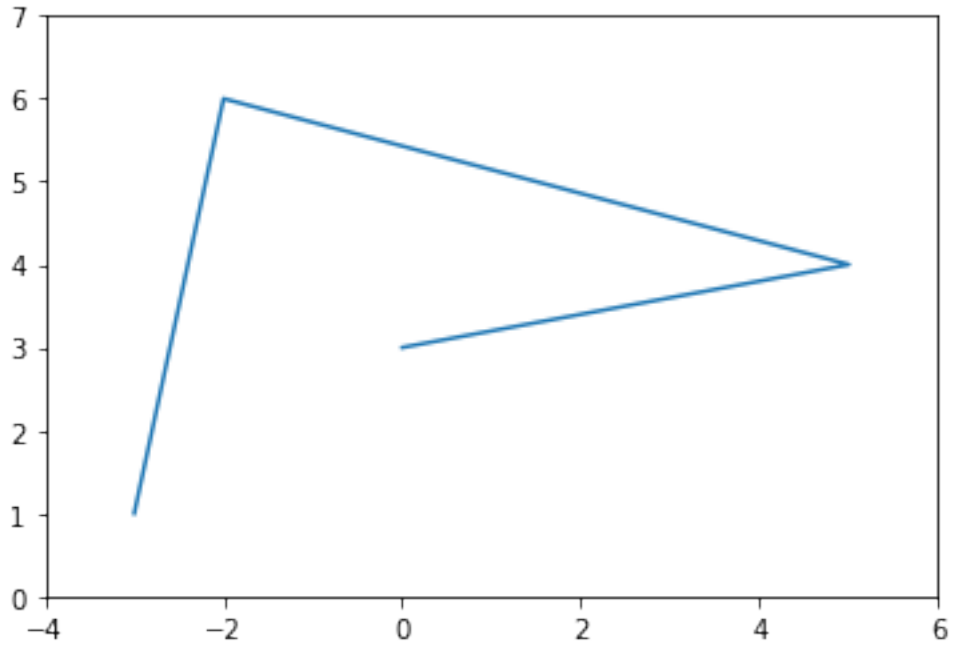
If the `plot` function is given one array of data, it will use it as the coordinates on the vertical axis, and it will just use each data point's index in the array as the horizontal coordinate. You can also provide two arrays: one for the horizontal axis `x`, and the second for the vertical axis `y`:

[ ]:
```
plt.plot([-3, -2, 5, 0], [1, 6, 4, 3])
plt.show()
```



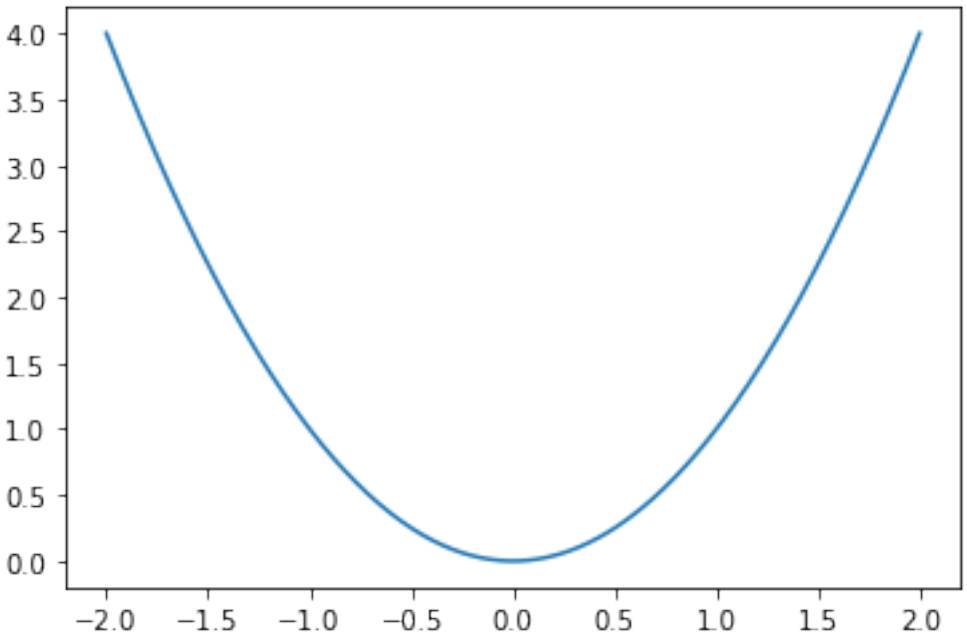
The axes automatically match the extent of the data. We would like to give the graph a bit more room, so let's call the `axis` function to change the extent of each axis `[xmin, xmax, ymin, ymax]`.

[ ]:
```
plt.plot([-3, -2, 5, 0], [1, 6, 4, 3])
plt.axis([-4, 6, 0, 7])
plt.show()
```



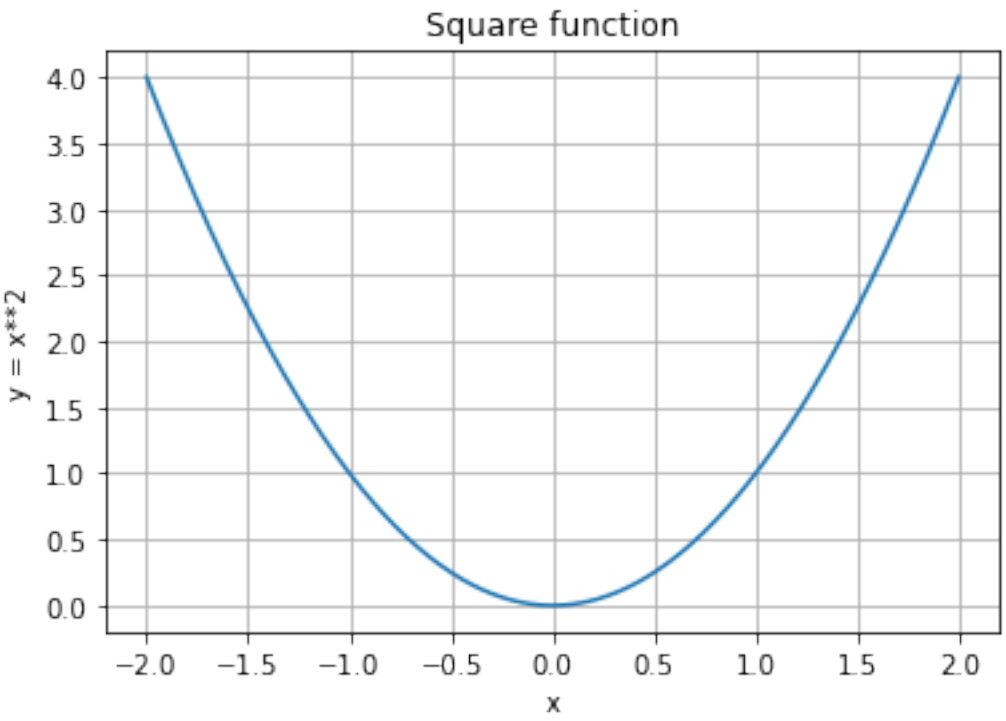
Now, let's plot a mathematical function. We use NumPy's `linspace` function to create an array `x` containing 500 floats ranging from -2 to 2, then we create a second array `y` computed as the square of `x` (to learn about NumPy, read the NumPy tutorial).

[ ]:
```
import numpy as np
x = np.linspace(-2, 2, 500)
y = x**2

plt.plot(x, y)
plt.show()
```



That's a bit dry, let's add a title, and x and y labels, and draw a grid.

[ ]:
```
plt.plot(x, y)
plt.title("Square function")
plt.xlabel("x")
plt.ylabel("y = x**2")
plt.grid(True)
plt.show()
```



You can pass a 3rd argument to change the line's style and color. For example `"g--"` means "green dashed line".

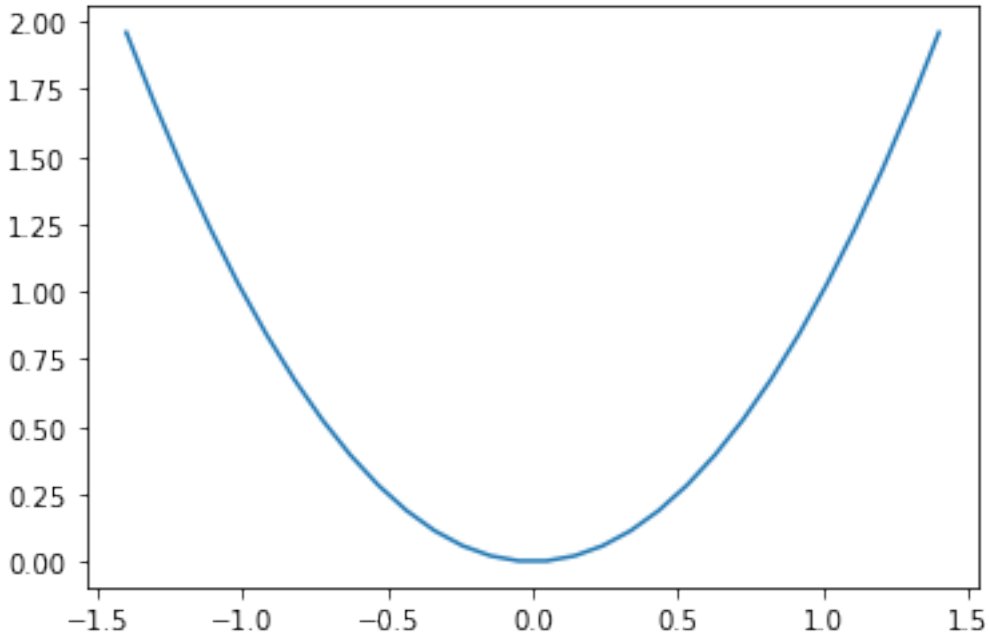
You can plot multiple lines on one graph very simply: just pass `x1, y1, [style1], x2, y2, [style2], ...`

For example:

19 Saving a figure

Saving a figure to disk is as simple as calling `savefig` with the name of the file (or a file object). The available image formats depend on the graphics backend you use.

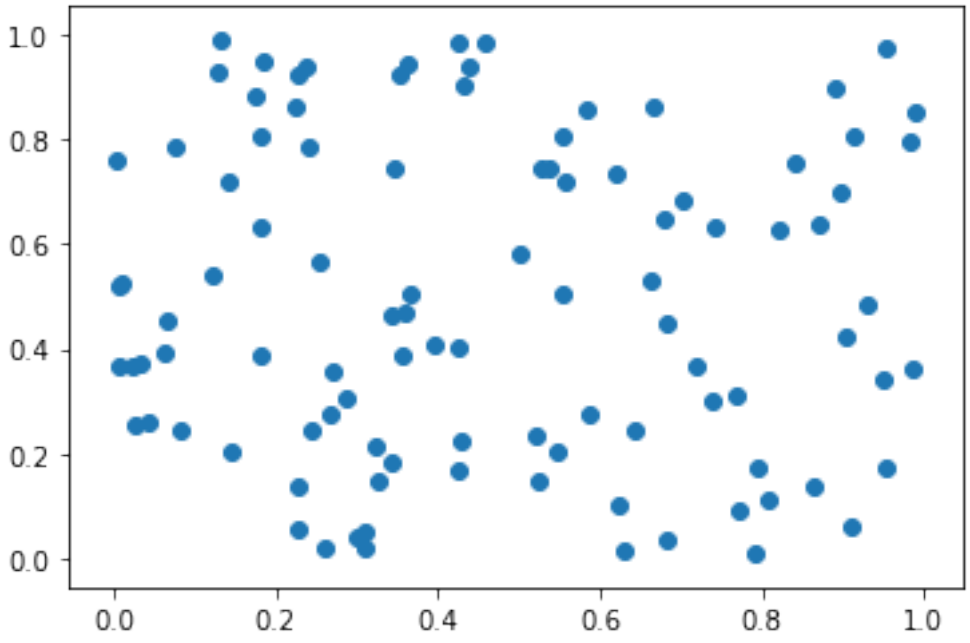
[ ]:
```
x = np.linspace(-1.4, 1.4, 30)
plt.plot(x, x**2)
plt.savefig("my_square_function.png", transparent=True)
```



20 Scatter plot

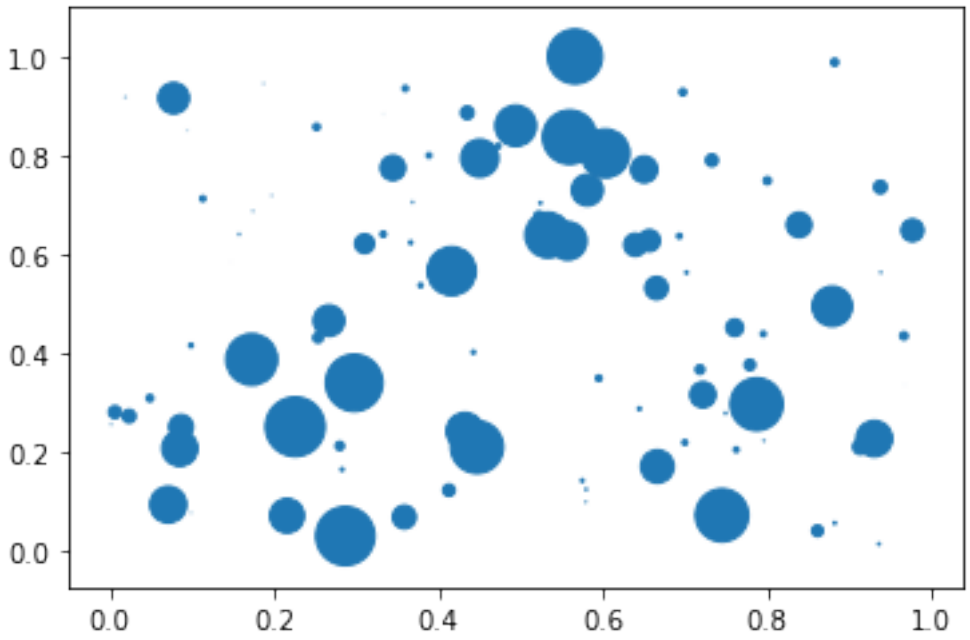
To draw a scatter plot, simply provide the x and y coordinates of the points.

[ ]:
```
from numpy.random import rand
x, y = rand(2, 100)
plt.scatter(x, y)
plt.show()
```



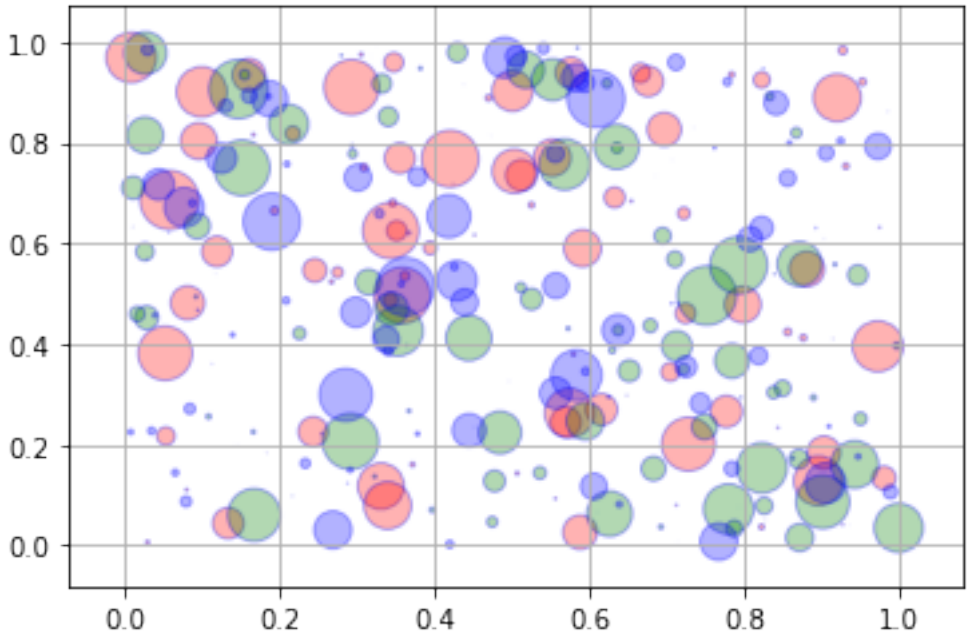
You may also optionally provide the scale of each point.

[ ]:
```
x, y, scale = rand(3, 100)
scale = 500 * scale ** 5
plt.scatter(x, y, s=scale)
plt.show()
```



And as usual there are a number of other attributes you can set, such as the fill and edge colors and the alpha level.

[ ]:
```
for color in ['red', 'green', 'blue']:
    n = 100
    x, y = rand(2, n)
    scale = 500.0 * rand(n) ** 5
    plt.scatter(x, y, s=scale, c=color, alpha=0.3, edgecolors='blue')

plt.grid(True)

plt.show()
```

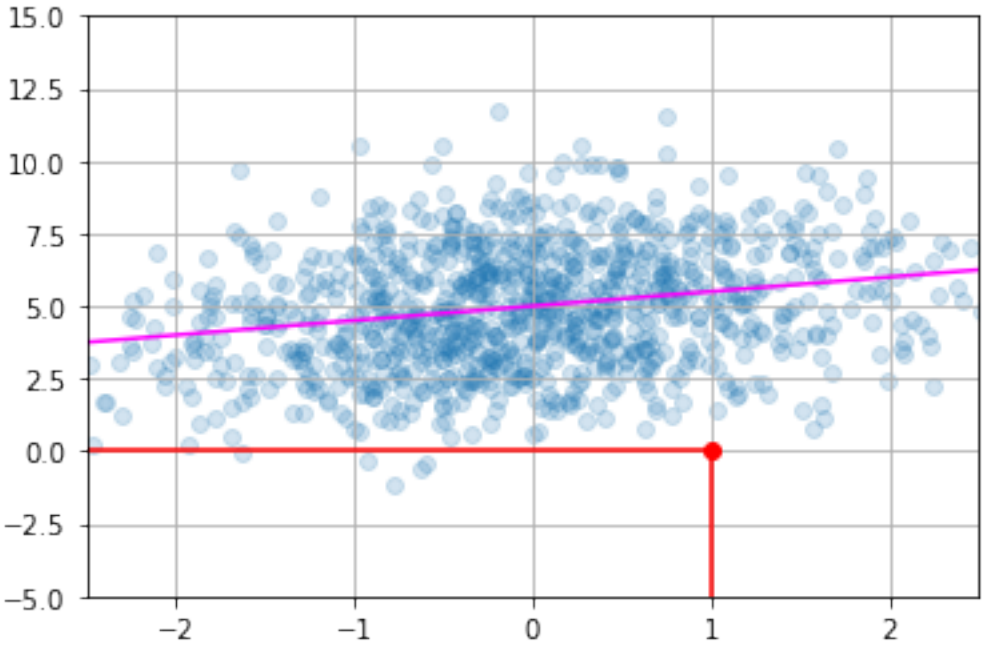


21 Lines

You can draw lines simply using the `plot` function, as we have done so far. However, it is often convenient to create a utility function that plots a (seemingly) infinite line across the graph, given a slope and an intercept. You can also use the `hlines` and `vlines` functions that plot horizontal and vertical line segments. For example:

[ ]:
```
from numpy.random import randn

def plot_line(axis, slope, intercept, **kargs):
```

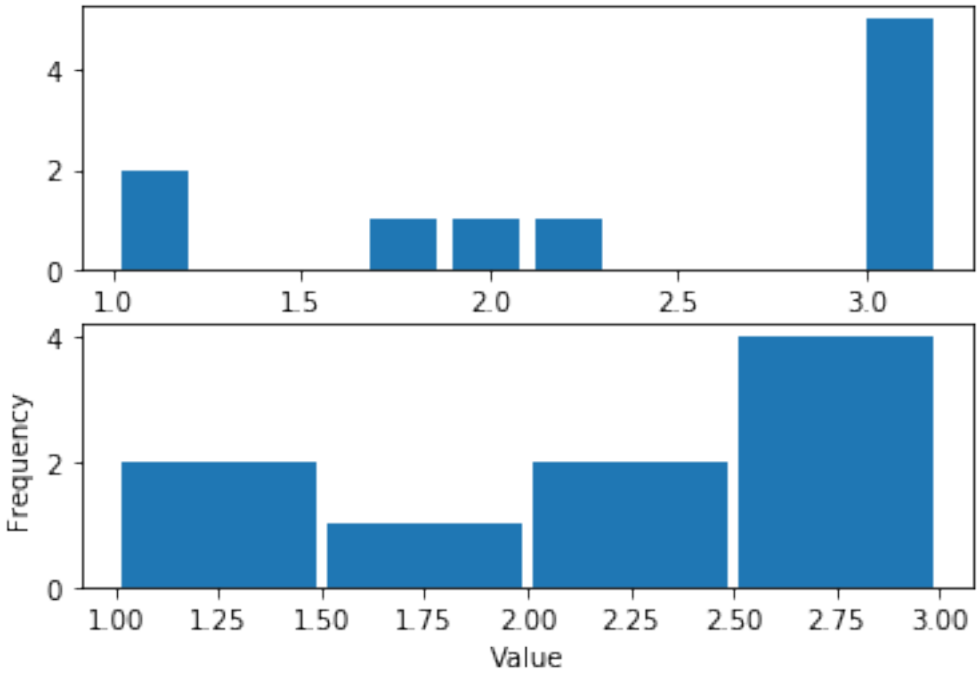
```
    xmin, xmax = axis.get_xlim()
    plt.plot([xmin, xmax], [xmin*slope+intercept, xmax*slope+intercept],␣
↪**kargs)

x = randn(1000)
y = 0.5*x + 5 + randn(1000)*2
plt.axis([-2.5, 2.5, -5, 15])
plt.scatter(x, y, alpha=0.2)
plt.plot(1, 0, "ro")
plt.vlines(1, -5, 0, color="red")
plt.hlines(0, -2.5, 1, color="red")
plot_line(axis=plt.gca(), slope=0.5, intercept=5, color="magenta")
plt.grid(True)
plt.show()
```



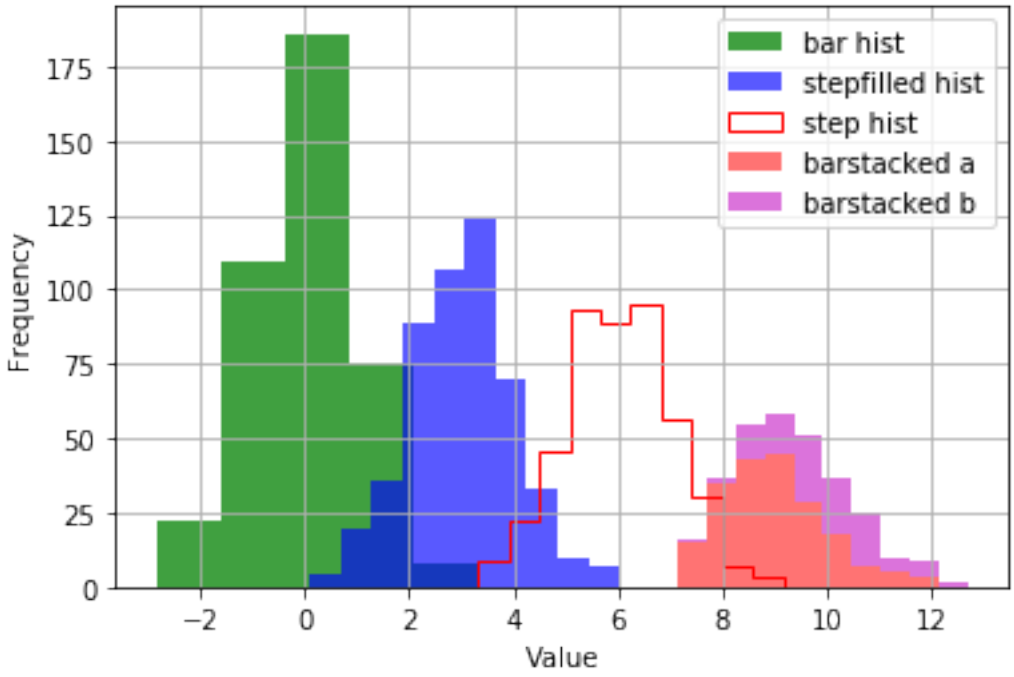
22 Histograms

[ ]:
```
data = [1, 1.1, 1.8, 2, 2.1, 3.2, 3, 3, 3, 3]
plt.subplot(211)
plt.hist(data, bins = 10, rwidth=0.8)

plt.subplot(212)
plt.hist(data, bins = [1, 1.5, 2, 2.5, 3], rwidth=0.95)
plt.xlabel("Value")
```

```
plt.ylabel("Frequency")

plt.show()
```



[ ]:
```
data1 = np.random.randn(400)
data2 = np.random.randn(500) + 3
data3 = np.random.randn(450) + 6
data4a = np.random.randn(200) + 9
data4b = np.random.randn(100) + 10

plt.hist(data1, bins=5, color='g', alpha=0.75, label='bar hist') # default␣
↪histtype='bar'
plt.hist(data2, color='b', alpha=0.65, histtype='stepfilled', label='stepfilled␣
↪hist')
plt.hist(data3, color='r', histtype='step', label='step hist')
plt.hist((data4a, data4b), color=('r','m'), alpha=0.55, histtype='barstacked',␣
↪label=('barstacked a', 'barstacked b'))

plt.xlabel("Value")
plt.ylabel("Frequency")
plt.legend()
plt.grid(True)
plt.show()
```


bar hist
stepfilled hist
step hist
barstacked a
barstacked b
Frequency
Value
0
25
50
75
100
125
150
175
−2
0
2
4
6
8
10
12