A BRIEF INTRODUCTION TO THE

PYTHON PROGRAMMING LANGUAGE

Ibrahim Elbadawy & Harvey Thompson

INTRODUCTION

Python was developed in the late 1980s and its name is inspired by the TV series, *Monty Python's Flying Circus*. Python is an open source, object-oriented language which is becoming popular for a range of scientific and engineering applications.

Unlike many other scientific languages, which are compiled into machine code for speed of execution, Python programs are run by an interpreter. This enables Python programs to be developed and debugged quickly but they can only be run on computers where the Python interpreter is installed. Python's main advantages are that it is open source (and therefore free), can be used with all the main operating sytems (Windows, Mac, OS, Linux, Unix etc.)

The Python interpreter can be downloaded from http://www.python.org/getit.

Python also comes with many editors, such as *Idle* or *Spyder,* and has excellent documentation easily available for its libraries such as the matplotlib library http://matplotlib.sourceforge.net/contents.html, or numpy, http://www.scipy.org/Numpy_Example_List.

Introductory Python programming concepts are now discussed briefly.

Excellent Introductory Texts on the use of Numerical Methods in Python include:

- Numerical Methods in Engineering with Python 3, Jaan Kiusalaas, Cambridge University Press, 2013.
- Python for Scientists, John M. Stewart, Cambridge University Press, 2017.

In the examples below the >>> indicates a Python command typed in and executed, as seen in the *Idle* editing environment.

There are a number of Python editor environments that can be used to run Python programs, such as *Idle* and *Spyder*. For example, if the *Idle* is used, you will see the interactive mode prompt >>>. A good way to start learning Python is to type commands into the prompt and see what happens! In *Idle*, batch mode, allows you to type in Python commands and save the commands into a Python program with the .py extension, e.g. *example_program.py*. You can then run the program by typing *example_program.py*. In Windows double-clicking on the program icon will also work. Rather annoyingly, the program window closes before you can read the results of running the program. You can stop this happening using the command within your program.

```
input('press return')
```

If you are using Unix or Linux and you specify the path to the Python interpreter in the first line of the program, you can also double-click the program icon. All programs start with the line `#!/use/bin/python` but note that on multi-user systems the path is usually `/usr/local/bin/python`.

A program can also be run from Idle using *Run/Run Module* menu.

When a module is loaded into a program for the first time with the import statement, it is compiled into a file with the extension .pyc, containg the bytecode of the module. The next time the program is run, the interpreter loads the bytecode and if changes have been made to the module, the module is automatically recompiled.

VARIABLES

A variable stores the value of a given type stored in a fixed memory location. In Python, unlike most other languages, the variable types may be changed. The following indicates that the Python syntax x=10 is typed in and executed and 10 is the result shown.

```
>>>x=10    # x is an integer type
10
>>>x=x*5.0   # now x is float type
50.0
```

The assignment x = 10 leads to x being associated with the integer value 10 while the second command associates with a floating point value of 50.0. Comments are indicated by the pound sign #.

ARITHMETIC OPERATORS

Python supports the arithmetic operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modular division

Examples:

```
>>>x=6-3
3
>>>x=18/3
6.0
>>>x=18/7
2.5714285714285716
>>>x=18.0/7
2.5714285714285716
>>>9%4
```

```
1   # (9 divided by 4 has remainder 1)
>>>x=6*7
42
>>>x=6*6*6
216
>>>6**3
216     # 6 to the power of 3
>>>5**12
244140625      # 5 to the power of 12
```

Python has augmented assignment operators, such as c+=d, as in C and C++. These are given below.

a+=b	a=a+b
a-=b	a=a-b
a*=b	a=a*b
a/=b	a=a/b
a**=b	a=a**b
a%=b	a=a%b

STRINGS

A string is a sequence of characters. They are enclosed by single ‘ ‘ or double quotes “ “.

```
>>>string1='Today is '
>>>string2='Monday'
>>>print(string1+' '+string2)    # add or concatenate the strings together
Today is Monday
print(string1[0:9])    # slices out the first 9 characters in string1
Today is
```

The **split** command splits a string into its component parts.

```
>>>s='29 08 1966'
>>>print(s.split())  # empty space (referred to as white space) is ignored
['29', '08', '1966']
```

A string is **immutable** meaning its type cannot be modified by an assignment statement. Trying to change the type of string leads to a TypeError.

```
>>>s='Today is Monday'
>>>s[0]='c'
```

```
Traceback (most recent call last):
  File "<ipython-input-10-5e0c096e0b49>", line 1, in <module>
    s[0]='c'
TypeError: 'str' object does not support item assignment
```

In some cases, there is a single or double quote in the sentence itself as follows

```
>>>'let's go fishing'
SyntaxError: invalid syntax
```

This gives an error. To avoid the error use a skip parameter (****) as follows

```
>>>"let\'s go fishing"
"let's go fishing"
```

Strings can be added together using (+)

```
>>>a=Muhammad"
>>>b=" Ali"
>>>print(a+b)
'Muhammad Ali'
```

Numbers cannot be added to strings because they are different types.

```
>>>num = 18
>>>"Some string" + num
TypeError: must be str, not int
```

To add these we need to convert a number to a string type by

```
>>>num =str(45)
>>>"I am " + num
I am 45
```

LISTS

A list is a sequence of objects separated by commas and enclosed in brackets. These are mutable in the sense that their contents can be changed by assignment operations. A list is a very powerful type for storing data in Python. For example if we define:

```
>>>Family = ['mom', 'dad', 'bro', 'sis', 'dog']
```

To get the various entries in the sequence:

```
>>>Family[0]
'mom'
>>>Family[4]
'dog'
```

You need to specify the index of the item you are interested in. Index '0' refers to the 1st item, '1' to the 2nd item, …,''n-1' to the nth item etc. You can also use backward indices from the end of the list in terms of negative indices:

```
>>>Family[-1]
'dog'
>>>Family[-5]
'mom'
```

Index '-1' refers to the end of the list etc. It is possible to slice out or extract pieces of a list. If the list is:

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

To extract numbers from the list you can use the syntax example[from number(included):to number (not included)]. For example:

```
>>>example[0:5]
```

[0, 1, 2, 3, 4]

```
>>>example[2:6]

[2, 3, 4, 5]

>>>example[:7]

[0, 1, 2, 3, 4, 5, 6]
```

Using the ':7' means take all elements from index 0 to index 7-1=6. All elements in the list can be obtained using

```
>>>example[:]

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

or, simply

```
>>>example

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

There are other ways of slicing the example list. example [starting number (included): ending number (not included): increment] displays the data with increments of increment.

```
>>>example[1:8:2]

[1, 3, 5, 7]
```

It is possible to add two lists together, for example

```
>>>[7,4,5]+[4,6,5]

[7, 4, 5, 4, 6, 5]
```

It is also possible to add strings together

```
>>>'Newcastle'+' United'
```

```
'Newcastle United'
```

but you cannot add two lists containing numbers and strings

```
>>>[4,5,6]+'Newcastle'
TypeError: can only concatenate list (not "str") to list
```

REPEATED STRINGS

`'string'*` followed by the number of repetitions

```
>>>'Hello'*4
'HelloHelloHelloHello '
```

Repeating numbers: `[number]*number of repetition`

```
>>>[21]*10
[21, 21, 21, 21, 21, 21, 21, 21, 21, 21]
```

But you cannot use

```
>>>21*10
210
```

because this is just simply a multiplication.

in: is a built-in key word in Python used to check that something exists in the list or not. If it exists the program returns a value of **true;** if it does not exist the program returns a value of **false**

```
>>>name='roastbeef'
>>>'z' in name
False
>>>'r' in name
True
>>>family=['mom','dad','bro']
>>>'sis' in family
False
>>>'bro' in family
True
```

LIST FUNCTIONS

These function can be used with sequences to give the minimum, maximum, list, specific number etc. in a sequence. Examples:

len( ): the length of list
max( ): the maximum number
min( ): the minimum number
list( ): make a list
list name [ ]= :used to change a list element to another value
del list name [ ]=: delete element from the list

For example:

```
>>>numbers=[8,1,4,17,28,165,7]

>>>len(numbers)

7

>>>max(numbers)

165

>>>min(numbers)

1

>>>list(numbers)

[8,1,4,17,28,165,7]
```

To change values in the list:

```
>>>numbers[3]=77

>>>numbers

[8, 1, 4, 77, 28, 165, 7]

>>>del numbers[3]

>>>numbers

[8, 1, 4, 28, 165, 7]
```

Slicing is often used to insert or delete elements in lists. To insert elements

```
>>>example=list('easyhoss')
>>>example[4:]=list('baby')
>>>example
['e', 'a', 's', 'y', 'b', 'a', 'b', 'y']
>>>example[4:]=list('racecars')
>>>example
['e', 'a', 's', 'y', 'r', 'a', 'c', 'e', 'c', 'a', 'r', 's']
```

To delete elements

```
>>>example=[7,3,3,3,8,9]
>>>example[1:5]=[]
>>>example
[7, 9]
```

To insert elements between existing ones:

```
>>>example=[7,8,9]
>>>example
[7, 8, 9]
>>>example[1:1]=[3,3,3]
>>>example
[7, 3, 3, 3, 8, 9]
```

Matrices can be represented as nested lists, with each row being an element of the list. Here is a 3x3 matrix d in the form of a list:

```
>>>d=[[1,2,3], \
              [4,5,6],  \
              [7,8,9]]
>>>print(d[1])              # print the second row (element 1)
[4, 5, 6]
>>>print(d[1][2])         # print third element of second row
6
```

In Python d[0] represents the first row, d[1] the second row etc. Note that it is usual to use *array objects* provided by the **numpy** module.

A **tuple** is a sequence of various objects separated by commas and enclosed in parentheses. Unlike lists, tuples are **immutable** and have the same operations as strings. Tuples with single objects require a final comma. For example y=(3,).

```
>>>info=('Black','White’,56)    # a tuple
>>>firstColour,secondColour,cost=info  # extract information from the tuple
>>>print(firstColour)
‘Black’
>>>print(info[1:3])
('White', 56)
```

Comparison operators return True or False. They are

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

```
>>>9 < 7
False
>>>9 <= 9
True
>>>9 != 3
True
>>>9 != 9
False

>>>one = [21,22,23]
>>>two = [21,22,23]
>>>one == two
True

>>>pizza = "pizzahut"
>>>'s' in pizza
False

>>>'p' in pizza
True
```

Conditionals

The if condition:

```
>>>If condition:

>>>     then execute block of code
```

executes a block of indented Python statements if the condition returns a True value. If the condition returns False the block is not executed. The if condition can be followed by any number of elif (short for 'else if') constructs

```
>>>elif  condition:

 >>>   then execute block of code
```

that works in the same manner. The else clause

```
>>>else:

>>>    then execute block of code
```

can be used to define the block of statements that are to be executed if none of the if-elif clauses are true. Examples:

Testing a true statement:

```
>>>tuna="fish"
>>>if tuna=="fish":
>>>       print ('this is a fish')
'this is a fish'
```

Testing a false statement:

```
>>>tuna="fish"
```

```
>>>if tuna=="bass":
>>>       print ('this is a fish')
```

Nothing is printed when this is executed.

```
>>>fish="tuna"
>>>if fish=="bass":
>>>    print ('this is bass alright')
>>>elif fish=='salmon':
>>>    print ('I hope i dont get salmonella poisining')
>>>elif fish=="trout":
>>>    print ('this is a trout')
>>>else:
>>>    print('I dont know what this is')
'I dont know what this is'
```

Nested Statements

```
>>>thing="animal"
>>>animal="cat"
>>>if thing == "animal":
>>>    if animal == "cat":
>>>        print ('this is a cat')
>>>    else:
>>>        print ('i dont know what this animal is')
>>>else:
>>>    print ('i dont know what this thing is')
‘this is a cat’
```

Another Example:

```
>>>thing="house"
>>>animal="cat"
>>>if thing == "animal":
>>>    if animal == "cat":
>>>        print ('this is a cat')
>>>    else:
```

```
>>>        print ('i dont know what this animal is')
>>>else:
>>>    print ('i dont know what this thing is')
'I dont know what this thing is'
```

Loops

The while construct

```
while condition:
    then execute block of code
```

executes a block of indented Python code if the condition is True. After execution of the block, the condition is tested once more. If it is still True, the block is executed again. This continues until the condition becomes False. The else clause can define the block of Python code to be executed if the condition is false.

Here is an example that creates the list of square numbers [1,4,9,16,25,36]:

```
N = 6
n=1
squares=[]     # create empty list
while n < N:
   squares.append(n*n)   # append element to the list
   n=n+1
print(squares)
```

This produces the output [1,4,9,16,25,36].

The for statement requires a target and a sequence over which the target loops. This takes the form:

```
for target in sequence
    then execute block of code
```

You may add an else clause that is executed after the for loop has finished. The previous program could be written with the for construct as

```
N=6
squares=[]
for n in range(1,N+1):
    squares.append(n*n)
print(squares)
```

Here n is the target and the range object is [1,2,…,N] created by calling the range function.

Loops can be terminated by a break statement. The following program searches for the first number in a list greater than 10.

```
list = [3, 5, 4, 11, 15, 6]
for i in range(len(list)):
    if (list[i]>10):
        print('list', i,'is > 10')
        break
    else:
        print('list', i,'is <= 10')
```

The results are:

```
list 0 is <= 10
list 1 is <= 10
list 2 is <= 10
list 3 is > 10
```

The continue statement enables a portion of an iterative loop to be skipped. If the interpreter executes a continue statement, it returns to the beginning of the loop without executing the statements that follow it. The following creates compiles a sub-list of numbers in a list that are greater than 10.

```
list = [3, 5, 4, 11, 15, 6]
sublist = []
for i in range(len(list)):
   if (list[i]<=10):
      continue
   else:
      sublist.append(list[i])
sublist
```

This creates the output `[11, 15]`.

Type conversion

Numbers are automatically converted to a common type before an operation on them is carried out. These include:

int(a)	Convert a to integer

float(a)	Convert a to float
complex(a)	Convert a to complex a+0i
complex(a,b)	Converts to complex a+bi

These can also be used to convert strings to numbers provided the string can be converted to a meaningful number. Examples

```
>>a=12

>>>b=4.8

>>>c='28.1'

>>>print(a+b)

16.8

>>>print(int(b))

4

>>>print(complex(a,b))

(12+4.8j)

>>>print(float(c))

28.1
```

Functions

The word **function** is a reserved word command in Python. Python supports the following mathematical functions

abs(a)	Absolute value of a
max(*sequence*)	Largest element of *sequence*
Min(*sequence*)	Smallest value of *sequence*
round(a,n)	Round a to n decimal places
cmp(a,b)	Returns $\begin{matrix} -1\ if\ a < b \\ 0\ if\ a = b \\ 1\ if\ a > b \end{matrix}$

However most mathematical functions are available in the **math** module

Examples: the exponent 5 to the power of 4 is written as 5**4 or one can use the power function, **pow** defined by pow(base number, exponent), pow(5,4). To get the absolute value of any number write abs (number). Examples are shown below

```
>>>5**4
625

>>>pow(5,4)
625

>>>abs(-18)
18

>>>abs(5)
```

```
5
```

If, however, we want to use the **floor** function to round to the nearest whole number, as in the following example, you need to import the math module first using the **import** keyword:

```
>>>import math

>>>math.floor(18.7)

18
```

To use the function **sqrt**, to find the square root of a number, need to write module name.function (parameter). For example:

```
>>>math.sqrt(81)
9.0
```

You can create a name for any function you want to use, as shown below:

```
>>>newname=math.sqrt

>>>newname(9)

3.0
```

>>>newname=math.floor

>>>newname(18.7)

18

This example makes the user input a name then creates an output string and prints it out:

```
>>>name = input("Enter name: ")

>>>output = "Hey " + name

>>>>print (output)

>>>Harvey <return>

'Hey Harvey'
```

This program will continue to accept names. However to make the program terminate after the first name we add input("Press<enter>") at the end.

>>>name = input("Enter name: ")

>>>output = "Hey " + name

>>>print (output)

>>>input("Press<enter>")

>>>Harvey <return>

'Hey Harvey'

The **print** function always puts a newline character at the end. We can replace the newline character with something else by using the keyword argument end. For example,

```
>>>print(f1,f2,…,end=' ')
```

replaces '\n' at the end with a space. Output can be formatted using the format method. Common format specifications are

wd	Integer
w.df	Floating point notation
w.de	Exponential notation

where w is the width of the field and d is the number of digits after the decimal point. Examples:

```
>>>x=345.37
m=45
>>>print('{:5.1f}'.format(x))
345.4
>>>print('m={:5d}'.format(m))    # pad with spaces
n=   45
>>>print('m={:05d}'.format(n))    # pad with zeros
n=00045
>>>print('{:8.2e} {:4d}'.format(x,m))
3.45e+02  45
```

Lambda Statements

If the function can be written simply in the form of an expression, it can be defined using a lambda statement:

func_name=lambda *param1, param2,…:expression*

Note: multiple statements are not allowed in lambda statements. For example:

```
>>>func=lambda x,y: 2*x**2+2*x-3*y**2
>>>print(func(3,4))
-24
```

Opening and Closing a File

Before you can read from or write to a file, you must create a file object with the command

file_object=open(*filename,action*)

where *filename* is a string that specifies the file to be opened (including its path if necessary) and *action* is one of the following strings:

'r'	Read from an existing file
'w'	Write to a file. If *filename* does not exist, it is created
'a'	Append to the end of the file
'r+'	Read and write to an existing file
'w+'	Same as 'r+' but *filename* is created if it does not exist.

‘a+’	Same as ‘w+’ but data is appended to the end of the file

It is good programming practice to close a file when access to it is no longer required. This can be done with the method

```
file_object.close()
```

Suppose we have a text file *example.txt* with the contents:

```
how now brown cow
```

We need to make sure that the editor we are using, for example *Idle* or *Sypder*, is pointing to the directory we think it is. To ensure this is the case you can use the following commands:

```
import os
os.chdir('c:\Python\Introduction')
print(os.getcwd() +'\n')
```

In this case, the Python codes being used are in the C:\Python\Introduction directory. You will need to change this to the directory you are using.

To read the first three characters use:

```
>>>fob=open('example.txt','r')
>>>fob.read(3)
'how'
```

To read all the characters use:

```
>>>fob=open('example.txt','r')
>>>fob.read()
'how now brown cow'
```

To read the first line in the *example.txt* file:

```
>>>fob=open('example.txt','r')
>>>print (fob.readline())
‘how now brown cow’
```

Suppose that *example.txt* now has three lines:

```
how now brown cow
one two three four
five six seven eight
```

To read all lines:

```
>>>fob.close()     # closes the file so that read goes back to the
beginning
>>>fob=open('example.txt','r')
```

```
>>>print(fob.readlines())
['how now brown cow\n', 'one two three four\n', 'five six seven eight']
```

Note that the '\n' is the newline character that terminates a line.

Writing data to text files:

```
>>>text1 = "Hello there\n"
>>>text2="how are you\n"
>>>text3="today?\n"
>>>textlines=text1+text2+text3
>>>fob=open("out.txt","w")
>>>fob.write(textlines)
>>>fob.close()
```

This creates the file *out.txt*:

```
Hello there
how are you
today?
```

Error Control

When an error occurs an exception is raised and the program stops. The **try** and **except** statements enable the conditions to be caught:

```
try:
    do one thing
except error:
    do another thing
```

The error is indicated by the name of a Python exception. If the error is not found then the **try** block is executed; however if the error is raised then the **except** block is executed instead. All exceptions can be caught by specifying an error in the **except** statement.

The following statement raises the exception *ZeroDivisionError*:

```
>>>x=-5.0/0.0
Traceback (most recent call last):
  File "<ipython-input-21-e808f469b946>", line 1, in <module>
    x=-5.0/0.0
ZeroDivisionError: float division by zero
```

This error can be caught by

```
try:
```

```
    x=-5.0/0.0
except ZeroDivisionError:
    print('You are dividing by zero')
```

Running this code results in:

```
You are dividing by zero
```

MODULES

It is good programming practice to store functions in modules, where the name of the module is the name of the file. A module can be loaded into a program by the statement:

```
>>>from module_name import *
```

This loads all functions available in *module_name*.

Python has many modules with functions for carrying out specific tasks.

The math module

Most mathematical functions in Python can be used through loading the **math** module. There are three ways of accessing the functions in a module. The statement. To load all of the functions in the math module type

```
>>>from math import *
```

This approach is not recommended as it can be wasteful and also lead to conflicts with other functions of the same name loaded in from different modules. It is better to be more specific about which modules you would like to import into your program using the syntax:

```
>>>from math import f1, f2,…
```

reads in functions f1, f2 etc from the math module. For example:

```
>>>import math
>>>math.sqrt(81)
9.0
```

You can also use an alias to access a module. For example

```
>>>import math as a
>>>a.sqrt(81)
9.0
```

To obtain a list of the functions in the math module use the following commands

```
>>>import math
>>>dir(math)
['__doc__',  '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan',
```

```
 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf',

 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh',

 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

The cmath module

This provides many of the functions in the math module, but with extensions so these can deal with complex numbers too.

```
>>>import cmath

>>>dir(cmath)

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh',

 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'inf', 'infj', 'isclose',
'isfinite', 'isinf', 'isnan', 'log', 'log10', 'nan',

 'nanj', 'phase', 'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'tau']
```

The numpy module

numpy introduces *array objects* that are useful matrix-type objects that can be used conveniently in several functions contained within it. This has very many functions and some of the most popular include

```
 ['complex' ,'float', 'abs', 'append', 'arccos', 'arccosh', 'arcsin',
'arcsinh', 'arctan', 'arctan2', 'arctanh',

'argmax', 'argmin', 'cos', 'cosh', 'diag', 'diagonal', 'dot', 'e', 'exp',
'floor', 'identity', 'inner', 'inv',

'log', 'log10', 'max', 'min', 'ones', 'outer', 'pi', 'prod', 'sin', 'sinh',
'size', 'solve', 'sqrt', sum', 'tan', 'zeros']
```

The array function creates an array from a list. For example:

```
>>>from numpy import array

>>>example=array([[8.0,7.0],[-5.0,7.0]])

>>>print(example)

[[ 8. 7.]

 [-5.  7.]]
```

Other useful functions include

```
zeros((size1,size2),type)
```

which creates a size1xsize2 array and fills it with zeroes of the specified type

```
ones((size1,size2),type)
```

which creates a size1xsize2 array and fills it with zerones of the specified type, where the default type is float.

The function `arange(start,end,increment)` returns an array, for example:

```
>>>from numpy import *
>>>print(arange(-5,4,3))
[-5 -2 1]
>>>print(arange(-6.0,9.0,3.0))
[-6. -3.  0.  3.  6.]
>>>print(zeros(5))
[ 0.  0.  0.  0.  0.]
```

For a rank-2 array, example say, the value in the ith row and jth column is given by example[I,j], remembering that the first row is row 0 and the first column is row 0. The values of the example array can be changed by a simple assignment operation.

```
>>>from numpy import *
>>>example=zeros((2,2),int)
>>>print(example)
[[0 0]
 [0 0]]
>>>example[0]=[-5,-10]
>>>example[1,0]=100
>>>print(example)
[[ -5 -10]
 [100   0]]
```

Arithmetic operators on arrays change each element in the array. For example

```
>>>from numpy import array
>>>example=array([5.0,10.0,30.0])
>>>print(example/2.5)
[  2.   4.  12.]
```

Mathematical functions on arrays also change each element in the array. For example

```
>>>from numpy import array,log10
>>>example=array([1.0,10.0,100.0])
```

```
>>>print(log10(example))
[ 0.  1.  2.]
```

numpy has many other useful functions that can perform operations on arrays. For example

```
>>>from numpy import *
>>>print(identity(4))
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
>>>example=array([[5,3],[3,1]],float)
>>> print(example)
[[ 5.  3.]
 [ 3.  1.]]
>>> print(diagonal(example))
[ 5.  1.]
```

The functions dot, inner and outer in numpy calculate array products very conveniently. For example to calculate the dot, or scalar, product of vectors and arrays we can use

```
>>>from numpy import *
>>>vect1=array([1,5])
>>>vect2=array([-3,11])
>>>array1=array([[0,1],[2,3]])
>>>array2=array([[4,5],[6,7]])
>>>print("dot(vect1,vect2)={0:d5}\n",dot(vect1,vect2))
>>>print("dot(array1,vect1)={0:d5}\n",dot(array1,vect1))
>>>print("dot(array1,array2)={0:d5}\n",dot(array1,array2))
dot(vect1,vect2)=
 52
dot(array1,vect1)=
 [ 5 17]
dot(array1,array2)=
 [[ 6  7]
 [26 31]]
```

numpy also has the linalg linear algebra module that can carry out important numerical tasks such as array inversion or solving systems of equations. For example

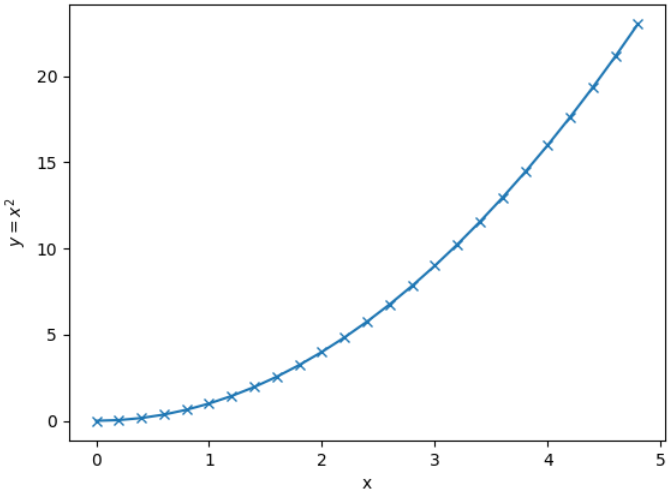
```
>>>from numpy import array
>>>from numpy.linalg import inv,solve
>>array1=array([[2.0,5.0,-1.0], \
         [1.0,1.0,1.0], \
         [0.0,1.0,2.0]])
>>>rhs=array([9.0,6.0,8.0])
>>>print(inv(array1))
[[-0.11111111  1.22222222 -0.66666667]
 [ 0.22222222 -0.44444444  0.33333333]
 [-0.11111111  0.22222222  0.33333333]]
>>>print(solve(array1,rhs))
[ 1.  2.  3.]
```

PLOTTING WITH MATPLOTLIB.PYPLOT

Python has access to MATLAB-style plotting functionality using the matplotlib.pyplot module. Here are some simple examples to show you how to plot out simple functions.

```
import matplotlib.pyplot as plt
from numpy import arange
x=arange(0.0,5.0,0.2)
y=x**2
plt.plot(x,y,'x-')     # plot with specified line and marker style
plt.xlabel('x')    # add label to x-axis
plt.ylabel('$y=x^2$')    # add label to x-axis
plt.savefig('test.png',format='png')    # save plot in png
plt.show()    # show plot on screen
```

This creates the following output:



Some of the line and marker styles that are possible are shown in the following table:

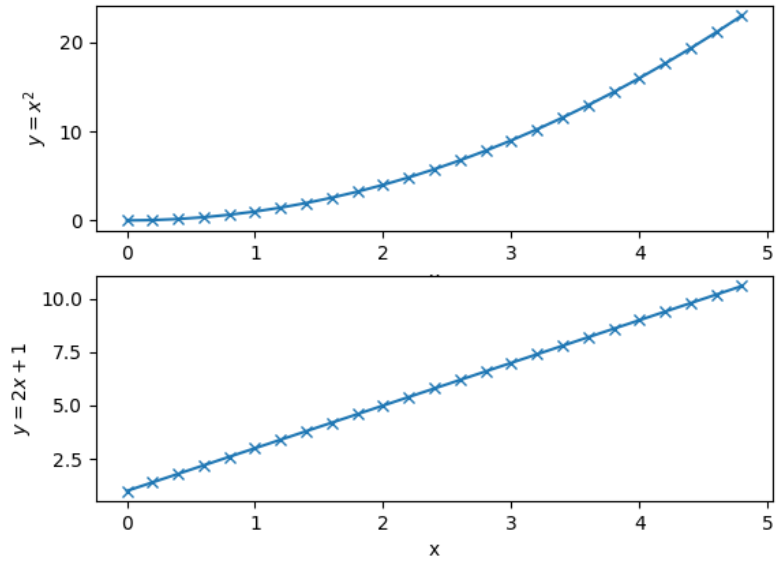
'-'	Solid line
'- -'	Dashed line
'-.'	Dash-dot line
':'	Dotted line
'o'	Circle marker
'^'	Triangle marker
's'	Square marker
'h'	Hexagon marker
'x'	x marker

The following figures shows how you can have more than one plot in a figure using the command subplot(*rows,cols,plot_number*). The parameters rows and cols divide the figure into a *rows x cols* grid of subplots. An example of two figures in vertical alignment:

```
import matplotlib.pyplot as plt
from numpy import arange
x=arange(0.0,5.0,0.2)
y1=x**2
y2=2*x+1
plt.subplot(2,1,1)
plt.plot(x,y1,'x-')     # plot with specified line and marker style
plt.xlabel('x')    # add label to x-axis
plt.ylabel('$y=x^2$')    # add label to x-axis
plt.subplot(2,1,2)
plt.plot(x,y2,'x-')     # plot with specified line and marker style
```

```
plt.xlabel('x')    # add label to x-axis
plt.ylabel('$y=2x+1$')    # add label to x-axis
plt.show()    # show plot on screen
```

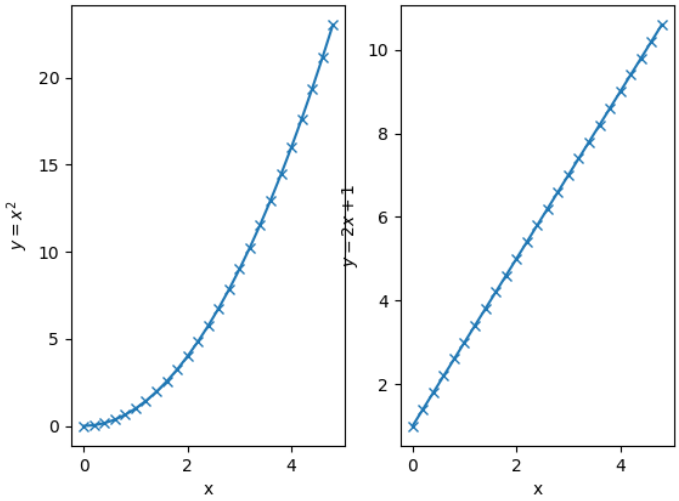
This creates the output



An example of figures in horizontal alignment:

```
import matplotlib.pyplot as plt
from numpy import arange
x=arange(0.0,5.0,0.2)
y1=x**2
y2=2*x+1
plt.subplot(1,2,1)
plt.plot(x,y1,'x-')     # plot with specified line and marker style
plt.xlabel('x')    # add label to x-axis
plt.ylabel('$y=x^2$')    # add label to x-axis
plt.subplot(1,2,2)
plt.plot(x,y2,'x-')     # plot with specified line and marker style
plt.xlabel('x')    # add label to x-axis
plt.ylabel('$y=2x+1$')    # add label to x-axis
plt.show()    # show plot on screen
```

This creates the output



SCOPING OF VARIABLES

Variables are defined and accessible within a Python program depending on the namespace they are defined in. A namespace is created and updated as a program runs and is a dictionary that contains the names of the variables and their values. Within Python there can be:

A **local namespace**, which is created when a function is called, with the variables passed to the function as arguments and the variables created within the function. It is deleted when the function terminates. It a variable is created inside a function, it cannot be accessed outside the function.

A **global** namespace - created when a module is loaded. Each module has its own namespace. All variables defined within a global namespace are visible to any function within that module.

A **built-in** namespace is created when Python runs and contains the core Python functions, which can be accessible anywhere within the Python program.

During a Python program execution the interpreter resolves the namespace by searching in the following order: local namespace, global namespace then built-in namespace. If the name cannot be resolved, Python raises a NameError exception. Although variables in the global namespace are visible to all functions within the module, it is recommended practice to pass them to functions which use them as arguments. These ideas can be illustrated in the following example code:

```
def func1():
     c=a**b
     print('a to the power of b=',c)


a=100.0
```

```
b=5.0

func1()

a to the power of b= 10000000000.0
```

In this case a and b have both been defined before func1() is called so they can be used to create the variable c, which is then printed out as shown. However, if the print statement is moved out of the function as follows:

```
def func1():
     c=a**b


a=100.0

b=5.0

func1()

print('a to the power of b=',c)
```

The we get the following error message since c has not been defined outside the function func1.

```
Traceback (most recent call last):


  File "<ipython-input-17-cbb468faf98f>", line 8, in <module>

    print('a to the power of b=',c)


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

OBJECT-ORIENTED PROGRAMMING IN PYTHON

An object is an entity that contains data, properties and methods (i.e. functions) associated with them. For example, person.name – person is the object, name is the property.

```
class exampleClass:
    eyes="blue"
    age=22
    def thisMethod(self):
        return ('hey this method worked')

>>>exampleObject=exampleClass()
>>>exampleObject.eyes
‘blue’

>>>exampleObject.age
22

>>>exampleObject.thisMethod()
```

'hey this method worked'

```
class className:

    def createName(self,name):

        self.name=name

    def displayName(self):

        return self.name

    def saying(self):

        print ("hello " + self.name)

>>>first=className()

>>>second=className()

>>>first.createName('Jim')

>>>second.createName('Tony')

>>>first.displayName()
```

'Jim'

```
>>>first.saying()
```

hello Jim

```
>>.first.name
```

'Jim'

Subclasses

Subclasses are used when they have parameters which already exist in the super class. For instance childClass (subclass) has the same parameters as in the parentClass (super class) as shown in the next example. Note that **pass** in the code does nothing.

```
class parentClass:
    var1="I am var1"
    var2="I am var2"

class childClass(parentClass):
    pass

>>>parentObject=parentClass()
>>>parentObject.var1
'I am var1'
>>>childObject=childClass()
>>>childObject.var1
'I am var1'
>>>childObject.var2
'I am var2'
```

Overwriting variables in a subclass

In this case, the subclass (child) has a different variable (var2=”toast”) instead of (var2=”sausage”). The example below shows how the variable, var2, is changed in the subclass.

```
class parent:
```

```
    var1="bacon"
    var2="sausage"

class child(parent):
    var2="toast"

>>>pob=parent()
>>>cob=child()
>>>pob.var1
'bacon'
>>>pob.var2
'sausage'
>>>cob.var1
'bacon'
>>>cob.var2
'toast'
```

Multiple Parent Classes

A class can include, or inherit, from many other classes. In the next example the child class inherits from the two classes of Mum and Dad and has its own variable, var3.

```
class Mum:
    var1="Hey i\'m Mum"

class Dad:
    var2="Hey there son I\'m Dad"

class child(Mum,Dad):
    var3="I\'m a new variable"

>>>childObject=child()
>>>childObject.var1
"Hey i'm Mum"

>>>childObject.var2
"Hey there son I'm Dad"

>>>childObject.var3
"I'm a new variable"
```

Constructors

```
class swine:

    def apples(self):

        print ("beef pie")

>>>obj=swine()

>>>obj.apples()

beef pie



class new:

    def __init__(self):

        print ("this is a constructor")
```

```
        print("this also prints out")
        var1=1.0
        print("var1= ",var1)
>>>newobj=new()
this is a constructor
this also prints out
var1=1.0
```

REFERENCES

Two excellent introductory references for scientific programming in Python are:

Jaan Kiusalaas, *Numerical Methods in Engineering with Python 3*, Cambridge University Press, 2013.

John M. Stewart, *Python for Scientists*, Cambridge University Press, 2017.