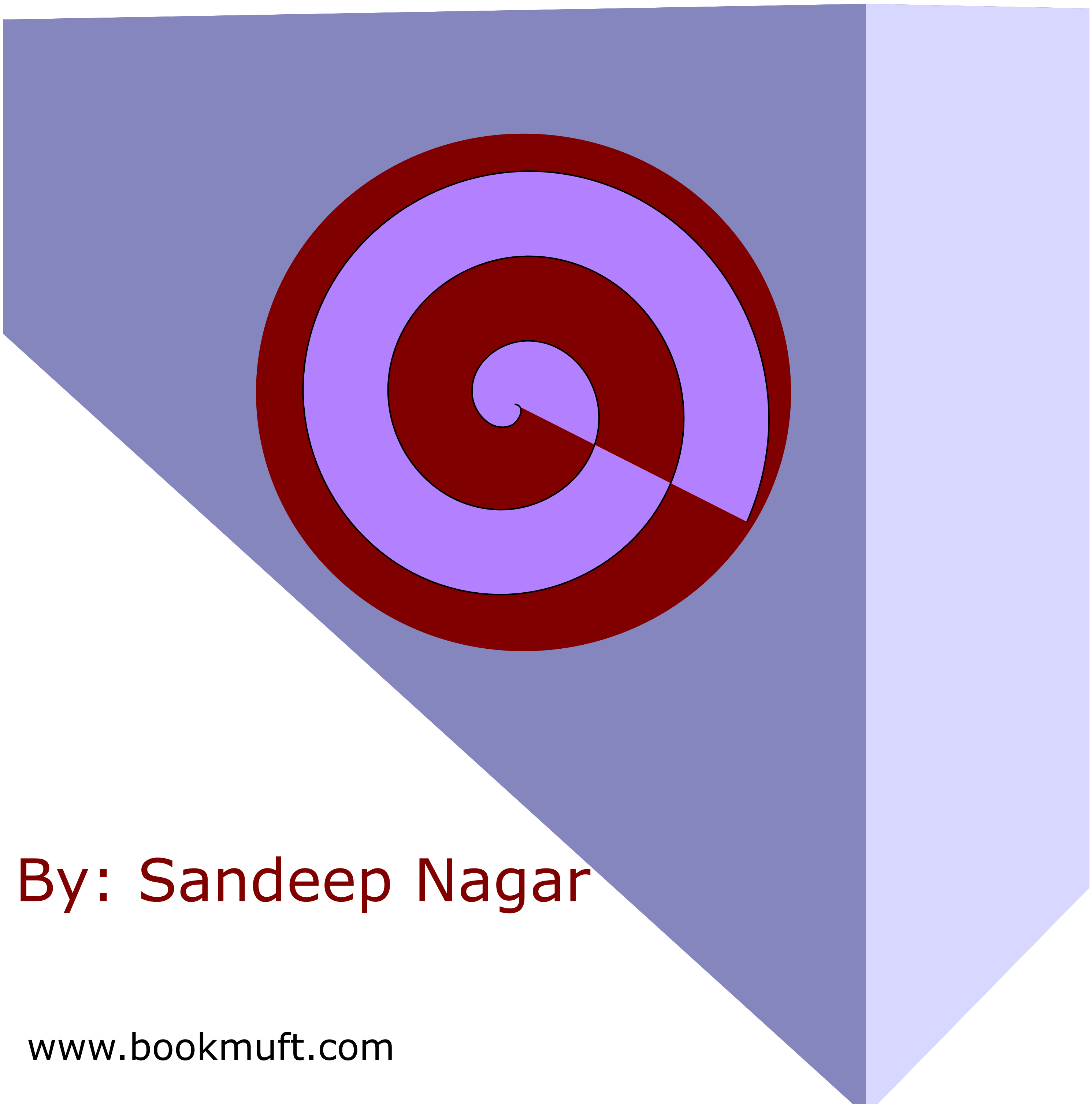
INTRODUCTION TO PYTHON

For Scientists and Engineers



By: Sandeep Nagar

Introduction to Python: For Scientists and Engineers

Dr. Sandeep Nagar

Tuesday 8th March, 2016

Contents

License information

Dedicated to two beautiful ladies in
my life, Rashmi and Aliya

1 Introduction to pythonic way of life

1.1 Introduction

Python emerged as a leader amongst well established and optimized languages like C, C++, Java for very simple reasons. Fabrication of python incorporates the philosophy that complex tasks can be done in simple ways. We tend to think that complex problems needs complex pathways to produce complex solutions. Python was fabricated with exactly opposite philosophy. Python was made to have an extremely flat learning curve and development process for software engineers. At the same time it was framed keeping in mind the power of Open Source movement, which helped in expanding its capabilities at amazing pace. Being open source in nature, people could make small programs and share amongst each other with ease. Group of programs to perform various tasks make up a module/package. There are over 57989 module till date (Tuesday 8th March, 2016), which has been submitted by equally large number of developers around the world. This made python jump rapidly amongst computer science community and finally grab number one position as the most favored programming language.

1.2 History

Development of python programming language dates back to 1980s. Its implementation was started by Guido van Rossum at CWI in the Netherlands in December 1989. This was an era when computing devices were becomes powerful and reliable every day. Python 1.0 was released to public in 1994, Python 2.0 in 2000 and Python 3.0 in 2008 but Python 3 was not made to be back compatible with Python 2 which made it less usable to users who were already developing with Python 2. This made a lot of developers stick to Python 2 even now with very few taker of Python 3 in general.

Python derives its philosophy from ABC language to a large extent. Synatx structure was largely derived from C and UNIX's Bourne shell environment served as inspiration for interpretative nature of the working environment. It also used a lot of other concepts from a variety of languages to make itself a robust multipurpose, object-oriented, high-level programming language. A high level programming language is the one which has strong abstraction from the details of computer. An object oriented programming language is the one which deals with data as an object on which different methods act, to produce a desired result. The abstract nature of objects makes it possible to invent objects of our choice and apply the programming concepts for a variety of applications.

1.3 Python and Engineering

Engineering problems employ numerical computations both at small scale and large scales. Hence the requirements of engineering application require a programming languages to fit well in both these regimes. There are very few languages which can boast these qualities and python is definitely a winner here. While running large computational tasks on bigger computational architectures, memory management, speed and reliability are the key parameters. Python being interpretative language is generally considered to be a slower options in this regard. But its ability to use faster codes written in C, Java and Fortran using interlinking packages `cython`, `jython` and `f2p`, speed intensive tasks can be run in native language within a python code. This relieved a lot of coders around the world who wondered if already optimized codes must be re-written in python.

Another aspect of engineering problems is ability of a programming languages to communicate with physical devices efficiently. Electronic devices are connected via wires, blue-tooth, wireless and Internet. Using an appropriate python module, one can connect to a compatible device to derive data from it and then visualize it in desired platform. A variety of micro-controllers (like Arduino) allow python to run its hardware with ease. Micro-computers like Raspberry Pi allow running python programs accessing the input-output devices. This enables cost effective prototyping of an engineering problem.

Users of $MATLAB^{®}$ argue that Simulink is one of the easiest way of prototyping and simulating a machine. Scilab also provide a similar platform called Xcos. Python still lacks this ability and budding programmers from coming generations can take this up as a challenge. A large community of developers are eagerly waiting for such a solution.

1.4 Modular programming

Modular nature of python programming incorporates the complex tasks being divided into small modules which seamlessly interact with each other. This enables both, development and debugging, easier. Modules can simply be imported to enables the use of various functions.

Python comes with thousands of modules to perform various tasks. Some of them are listed in table 1.1

Package name	**Meaning**	**Purpose**
`numpy`	numerical python	Numerical computation
`scipy`	Scientific python	Scientific computations
`sympy`	Symbolic python	Symbolic computing
`matplotlib`	Mathematical Plotting Library	For plotting graphs

Table 1.1: Python modules

There are thousands of packages available for download at website for **python package index** `https://pypi.python.org/pypi/pip`. Installing packages can be quite tedious job when one needs to install them in proper directories and assign the installation paths at proper places. To make life simple for developer working with ubuntu, one simply needs to write

```
$ sudo pip install numpy
```

Above command install `numpy` at proper place. Similarly, other packages should be installed as and when required. It is important to note that whereas modern day personal computer (PC) offers large memory to use, micro-computers like Raspberry Pi has limited memory. Hence judicious use of these memory resources is highly recommended. Since all modules occupy some memory, hence they should be installed on *need-to-install* basis. Also they should be imported in the program as and when required. Python allows selective import if specific functions to optimize memory usage. It is considered a good practice to write programs which avoid wasteful use of resources.

Mentioning use of each modules is beyond the scope of present book. Modules will be introduced as per requirement of the topic. User is encourage of check various modules and their documentation for usage. A general use of modules and their function will be dealt at later point in present book.

1.5 summary

Python has gained a lot of attention world-wide owing to its flat learning and steep development curves. It has gained number one spot in recent times in terms of popularity and choice of programming language. Owing to a large base of developers due to open-source model, it has a rich library of modules for various tasks required to solve an engineering problem at hand. Hence python educated engineers can fulfill the demands of modern industry which demands fast and efficient solutions to their problems.

2 Introduction to basics of python

2.1 Introduction to python as an interpreted language

Python is an interpreted language as opposed to compiled languages like C, C++, Java etc. Each line of code is interpreted and executed one by one, as per their order. This makes the architecture of computation quite different than traditional languages. For example, suppose line 5 of a python program has syntax error, in this case the program will executes all commands till line 4 and will then show an error. In case of compiled languages, the compilation steps would show error and program will not run at all. To understand this difference elaborately one needs to understand the processes of compilation and interpretation.

In the case of compiled languages, a compiler translates the *human readable* code into *machine readable* assembly language. Machine readable code is are called *object code* given by object files. These object files can be run directly on machines. As an example lets assume that code is given as:

```
/* Hello World program */

```

```
#include<stdio.h>

main()
{
printf("Hello World");

}
```

Suppose this code is saved as `hello.c`. To compile this code on UNIX like machine with `gcc` compiler, we give command as:

```
gcc hello.c -o hello
```

This creates an object code named `hello`. During compilation, the header `stdio.h` is used to understand the input-output statements such as `printf("Hello World")`.The object code can then be executed by writing on UNIX terminal:

```
./hello
```

The object file can be shared by the user with anybody and if the microprocessor architecture is same as that of user then, it will be executed uniformly.

But this is not the case with python. Being interpreted language, it employs an *interpreter* which interprets the code into an intermediate code and then to machine code. An interpreter reads the source text of a program, analyzes it, and executes it one line at a time. This process is very slow as the interpreter spends a lot of time in analyzing strings of characters to figure out what they mean. For example, to type **hello world** as done by above C program, a python program will simply require:

```
print "hello world"
```

In just one line, an interpreter scans the world *print* and looks for what it means. In python interpreter, it means to print to a particular device. A device can be a computer terminal, printer plotter etc. By default, its a computer terminal. Print commands also demands *arguments* which is scanned in second step as a string **hello world** (A string in python can be enclosed in " or ""). Hence the complete interpretation of the line is to print the string `hello world` on a computer terminal.

When the programs compose of hundreds and thousands of lines, a compilation process will yield a faster result because the object code needs to be only compiled once and then run directly on microprocessor. Whereas an interpreted code will check for *interpretations* each time it needs to be processed.

Despite these odds of being inherently slow, it has becomes favorite amongst scientists and engineers for being extremely simple, intuitive and powerful due to rich library of modules for various computational tasks. Present chapter will discuss some of them in detail.

2.2 Installation

To work with python, it must be installed first. Present book is written using Ubuntu 14.04 system where python comes per-installed. In case of other systems, user is advised to visit (`https://www.python.org/downloads/`) and download python 2.x where x shows the version number of python. Users who wish to work in Integrated Development Environment (IDEs) needs to explore the website at (`http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Python`). Canopy is quite convenient python IDE and its academic version is free for students and teachers at an educational institution. User is free to install any one of them and run python commands at command prompt or save a script file with `filename.py` extension and then run the command `python filename.py` at the command prompt.

2.3 Python as a calculator

Python prompt can be used as a calculator in its simplest form. On the python command prompt, following commands can be checked for $2 + 4$

```
>2+4
>6

>2+4.
```

```
>6.0
```

As per calculations above, $2 + 4$ yields 6 whereas $2 + 4.$ yields 6.0. 6 and 6.0 are two different objects for a computer. 6 is an *integer* stored in lesser space than 6.0, which is a *floating point number*. Just like two types of numerical data, data can be classified in different kinds of objects (need not be numeric). Python interprets everything as an object. An object, when defined, needs to be defined with its attributes/properties. For example, a floating point number has different rules of addition, substation, printing on screen, representation of graphs etc, when compared to an integer. Hence a floating point number is quite different **data type** when compared to an integer. A detailed list of data types will be discussed later.

It is also important to note than to define a floating point number 4.0, writing even `4.` is sufficient. `4.0` and `4.` are equivalent. similarly `0.4` and `.4` both mean the mathematical number 0.4. Next two chapters will discuss various types of data and various operators that can operate on these data types.

2.4 Modules

Python multi-verse has expanded with thousands of modules and being open source, most of them are readily available too. Modules are collection of python programs to accomplish specific tasks. For example, `numpy` has various facilities for numerical computation which was further expanded into `scipy` for scientific computation in general. `matplotlib` is acronym for mathematical plotting library, which has rich features to plot a variety of publication-ready graphs. `pandas` is the library for data analysis, `scikit-learn` for machine learning, `scikit-image` for image processing `sympy` for symbolic computing etc.

To use a module, it must be installed in the machine first. Installation includes downloading the files properly into an appropriate folder or directory, unzipping it and defining proper paths. There is an easier way for Ubuntu users where a short command line base program `pip` performs these tasks seamlessly.

Running a simple command:

```
sudo apt-get install python-pip
```

installs the program `pip` first. It can then be used to install a package say `numpy` by simply issuing the command

```
pip install numpy
```

Replacing the name of package with the desired package will simply do the trick of installing the packages hassle free.

Installing **scipy stack** is most useful for the present book because it install a variety of programs which will be used henceforth. It can be installed by issuing the command on an Ubuntu terminal:

```
   $ sudo apt-get install python-numpy python-scipy
python-matplotlib ipython ipython-notebook
python-pandas python-sympy python-nose
```

Above command in a single line installs `numpy, scipy, matplotlib`, `ipython, ipython-notebook, pandas, sympy, nose`.

2.4.1 Using a module

To use a particular module and its functionalities, one must first import it inside the workspace.

2.5 Summary

Python has an extremely flat learning curves owning to the fact that its interpretive language due to which can be insert instructions line by line and run them subsequently. This methods avoids compilation and subsequent errors which prove to be major stumbling block for a beginner, who has limited knowledge of the inner workings of the programming language.

3 Data types

3.1 Introduction to Various types of data

Modern computers distinguish data as various types. Data can be numbers, characters, strings (a group of characters) etc. In python, one can defines new data types as and when required. There as some built-in data types for handling numbers and characters. Different data types occupy different amount of memory. It is judicious to understand the needs and choose a data type accordingly. Data type also determines the accuracy of the answer. Following sections will discuss various built-in data types in python.

3.2 Logical

logical: This type of data stores boolean values `True` or `False` boolean values and can be operated by boolean operators like `AND`, `OR` etc. Most programming languages use the values `1` or `0` for boolean values but python differs in this approach.

3.3 Numeric

Numeric: There are four types of numeric data types:

- *int*: Integers
- *long*: long integers
- *float*: Floating point numbers
- *complex*: complex numbers

3.3.1 Integer

Python has arbitrary precision for `float, long, complex`, hence the limit to length of these numbers is subject to availability of memory. The positive side of this architecture is that one is not limited to a range of numbers. But one must always ensure that sufficient memory is available during the calculation, to avoid erroneous results. Python 2 limits the size of `int` to bytes (to same size as C programming language), whereas Python 3 has merged `int` and `long` as `int`. On a 32-bit system, Python 2 stores `int` as 32 bits. The range of integers can be obtained using the module `sys`, whose function `maxint` returns the value of maximum values of integer stored by python.

```
>>>import sys
>>>sys.maxint
2147483647
>>>sys.maxint+1
2147483648L
>>>-1-sys.maxint
-2147483648
```

As shown above the maximum value for integer for a 32 bt machine is 2147483647. When this value is incremented by 1, it is automatically upgraded to a `long` type (which stores numbers with arbitrary precision). This is indicated by L mentioned after writing the number :`2147483648L`. The minimum value of integer is given by `-1-sys.maxint` which is given as −2147483648.

Python doesn't have built-in unsigned types for integers as with some other programming languages like C. To make a negative number positive, one can simply use `abs()` function like `abs(-1)` is obtained as `1`.

The in-built function `type()` presents the data type as follows:

```
>>>type(-1)
int

>>>type(1)
int

>>>type(sys.maxint+1)
long
```

As seen in above examples, python interpreter allocates data type dynamically i.e. it allocates the data type to any integer more than 2147483647 as `long` without any user input. This is quite convenient for a programmer.

It is worth mentioning that small numbers can be stored as `long`. `0L` is different than `)` as it is stored as `long` types as opposed to an `int` type. From memory usage point of view, integers should be used as integers, wherever required. `long` are stored in bigger memory space and also consume more time while processing.

```
>>>type(0L)
long

>>>type(0)
int

>>>print sys.getsizeof
    (10000000000000000000000000000000000000000000000)
48
```

Function `sys.getsizeof()` in module `sys` gives the number of bytes required to store a particular number. This is a handy function to know and control the numerical data storage.

3.3.2 Floating point numbers

In computing, **floating point** notation is a scheme of representing an approximation of a mathematical real number. This scheme can trade-off between range and precision. A real number is usually written with a decimal point. For example, 2 is an integer whereas 2.0 is a real number. These two numbers are quite different for a computer. Whereas 2 will be stored as `int` type, 2.0 will be stored as `float` type.

```
>>>type(2)
int

>>>type(2.0)
float
```

The issue with floating point number based arithmetic is that the answer is an approximation of real number since real numbers are defined for 10 as their base whereas computer works with numbers where 2 is used as the base. For example: 0.123 is defined as

$$0.123 \rightarrow \frac{1}{10^1} + \frac{2}{10^2} + \frac{3}{10^3}$$

in the number system with base 10 whereas in number system with base 2, it is represented as

$$0.123 \rightarrow \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3}$$

Above calculation shows that $0.123_2 = 0.135_{10}$. If one uses more number of bits to store the value, one gets a better approximation of the real number, but one is always limited to use **approximated** values instead of **real** values.

```
>>>from decimal import Decimal
>>>Decimal(0.123)
Decimal('0.1229999999999999982236431605997495353221893310546875'
    )
>>>Decimal(1.2345)
Decimal('1.2344999999999999307220832633902318775653839111328125'
    )
>>>type(Decimal(0.123))
Out[39]: decimal.Decimal
```

One can use the `decimal` module which has a function `Decimal()` that returns the number as stored by the computer. As seen above 0.123 is stored as 0.1229999999999999982236431605997495353221893310546875 which is still an approximation of the real number 0.123. For most cases, the error is insignificant and one can ignore the fact that calculations using digital computer (running on binary system of number) has introduced some error. But for some cases, this error is significant and one must take proper measures to calculate this error and counter the same.

```
>>>sys.getsizeof(Decimal(0.123))
72
>>>sys.getsizeof(0.123)
24
>>>print "%e" %(Decimal(0.123))
1.230000e-01
>>>print "%f" %(0.123)
0.123000
```

As seen above, `Decimal(0.123)` occupies 72 bits and hence is more accurate approximation as compared to `0.123` which occupies 24 bits. Also floating point numbers can be printed as numbers with decimal points using formating argument `%f` or they can also be printed in scientific notation as 1.23×10^{-1} using formatting argument `%e`.

3.3.3 complex Numbers

Complex numbers are extensively used in science and engineering studies. The imaginary part of a complex number has important information about phase of a signal. Python enables the use of complex numbers by creating an object called `complex`.

```
>>>type(2+3j)
complex
>>>sys.getsizeof(2)
24
>>>sys.getsizeof(2+3j)
32
>>>complex(2,3)
```

```
(2+3j)
```

As seen above a complex number is defined in two parts: `a+bj` where `a` represents the real part, `b` represents the imaginary part and `j`$=\sqrt{-1}$. They occupy more space in memory, to accommodate the information about real and imaginary parts. `complex(x,y)` is another way of defining a complex number where an *ordered pair* of two real number is defined, which make up real and imaginary part. Here the ordering is important as `x` makes the real part and `y` makes the imaginary part.

3.4 Sequences

Any symbol which requires storage is known as a character. Everything that appears on computer screen and printed papers, is considered character in programming languages. This includes ASCII and extended ASCII characters. Examples of characters include letters, numeral digits, whitespace, punctuation marks, exclamation mark etc. In general, all keys on keyboard, produce characters.

For the purpose of communicating between computing devices, characters are encoded in well defined internationally accepted formats (like ASCII,UTF-8 etc) which assigns each character to a string of binary numbers. Two examples of usual encodings are ASCII and the UTF-8 encoding for Unicode. All programming languages must be able to decode and handle internationally accepted characters. Python also deal with characters using data type `string, list, tuple`.

3.4.1 String

A string is simply a sequence of 8 bit characters. Lower case and upper case characters has different encoding hence strings are case sensitive.

```
>>>type('a')
str
>>>type('abba')
str
>>>type("a")
str
>>>type("abba")
```

```
str
```

Here we defined a string of one and four characters respectively. It is important to remember that white-space is also a character. Hence, `Hello world!` has 12 characters namely `h,e,l,l,o, ,w,o,r,l,d,!`. While defining strings, we enclose the characters under `''` or `""`. When a string has to span in multiple lines than triple quotes are used like:

```
>>>print """"Python is an interpretive language
...It is one of the finest ones in its category"""
Python is an interpretive language
It is one of the finest ones in its category
```

3.4.2 list and tuples

A `list` is simply a group of objects, irrespective of its data type.

```
>>>type(('a',1,2.0,3+4j))
tuple

>>>type(['a',1,2.0,3+4j])
list
```

The only difference in thier definitions is the type of brackets enclosing them: list is defined with `[]` brackets and tuple is defined with `()` brackets. As seen above a tuple and list is defined with a string, integer, float and complex number data type as its **elements**. The only difference between list and tuples are that tuples are immutable lists i.e. their elements, once defined, cannot be altered. Elements of a list can be altered using their indices. More information about how this is done is given in next chapter where operations on lists have been defined. In scientific computing, universal constants can be defined as a tuple and then can be accessed in a program using its index.

3.5 Set and Frozen Set

The `set` data type is implementation of mathematical set. It is an **unordered** collection of objects. Unlike sequence objects like `list` and `tuple`, where elements are ordered, sets do not have such requirements. Sets do not permit duplicity in occurrence of an element, i.e an element wither exist 0 or 1 times.

```
>>>set(['h','e','l','l','o',1,2.0,3+4j])
{1, 2.0, 'e', 'h', 'l', 'o', (3+4j)}
>>>frozenset(['h',0,1.0,2+3j])
frozenset({0, 'h', (2+3j), 1.0})
```

Please note that an since `l` occurred two times while defining the set, it was gives only one membership. Set operations are discussed in detail in subsequent chapter. A **frozen set** is simply immutable set.

3.6 Mappings

Mapping is a scheme of defining data where each element is identified with a key called "hash tag". The element can be accessed by referring to the key. One of the data type in this category is a **dictionary**.

A dictionary is an *unordered* pair of values associated with keys. These values are accessed with keys instead of index. These keys have to be hashable like integers, floating point numbers, strings, tuples, and frozensets. Lists, dictionaries, and sets other than frozensets are not hashable. An example of a dictionary is given below:

```
>>>dict{'a':1, 'b':10}
{'a': 1, 'b': 10}

>>>'a' in dict([("a", 1), ("b", 10)])
True

>>>'b' in dict([("a", 1), ("b", 10)])
True

>>>'c' in dict([("a", 1), ("b", 10)])
False
```

In example above, we created a dictionary containing two characters `a` and `b` identified by two keys `1` and `10`. Subsequent commands checks if characters `a`, `b`, `c` are past of the dictionary to which we get an answer `True` for `a` and `b` but `False` for `c`. A variety of operators can operate on dictionaries as discussed in subsequent chapter.

3.7 Null object

`None` is a null object. Null objects in other programming languages like C, Java, PhP are given by the keyword `null`, but in python its is denoted by the keyword `None`. It refers to non-functionality i.e. no behavior for the object with which it is associated. It is used in cases where we wish to perform an action which may or may not work. Using `None`, one can check the state of action at later point. `help(None)` and `http://www.pythoncentral.io/python-null-equivalent-none/` gives useful insights in its use.

3.8 Summary

Object oriented programming uses the fact that all computing entities are merely objects which *interact* with each other as per their defined behavior. Some built-in data types have been discussed in present chapter. Some data types are defined inside the modules. One can define ones own data types and define its properties. Before going to these advanced topics, ti will be useful to know how operators operate on various kinds of data. This will be subject of next chapter.

4 Operators

4.1 Introduction

Operators work in similar fashion as mathematical functions. They provide a relationship between two different domains. For example, multiplication operator makes an ordered pair of operands (data on which operator works) and produce another data point. This can be done to any number of data points. In this way, operator transforms data from domain of operands to domain of results.

In the domain of numerical calculations, one needs to use basic and complex mathematical functions/operators like multiplication (`*`), addition (`+`), subtraction (`-`), division (`/`), Modulus (`%`), Exponentiation (**) etc. These operators can be combined in complex manner to perform an arithmetic operation. Depending on data type, they define thier functionality. For example, on numeric data `+` performs numeric addition where as on a string it will perform concatenation.

```
>>> 2+3
5
>>> 'a' + 'b'
```

```
'ab'
>>> "hello" + " " + "world" + "!"
'hello world !
```

There are a variety of operators which can operate on data types as discussed in chapter 3.

Following section will discuss a variety of built-in operators like arithmetic, logical/boolean etc. Please note that the field of operators is not limited to discussion in present chapter. Modules define new data types for which new operators are defined. We shall confine our discussion to built-in basic operators only.

4.2 Concept of variables

Due to operators acting on data, its values can change during the course of computation. To store values temporarily (i.e. during the course of computation), we use variables. Variables store a particular value at a memory location and address it with a symbol or set of symbols (called *strings*). For example: one can store the value of 0.12 as a variable `a` and then use it in an equation like

$$a^2 + 10 \times a$$

.

```
>>>a=0.12
>>>answer = (a**2) + (10 * a)
>>>print answer
1.2144
>>>type(a)
float
```

Here the numerical value 0.12 is stored at a memory location known by the name `a` and this is called by subsequent equation defined using a variable name `answer`, which when printed, prints the value stored in it. The values stored is *floating point number*. The type of object can be known by a function `type()` which takes the variable name as its argument.

Python variables needs not be explicitly defined for their *type*.

4.2.1 Rules of naming variables

- Must begin with a letter (a - z, A - B) or underscore (_)
- rest of the characters can be letters, numbers or _
- Names are case sensitive
- There is no limit to length of names but its wise to keep them short and meaningful
- `keywords` cannot be used as variable names.

Using the module `keyword`, one can obtain the list of keywords using the function `keyword.kwlist`. The code given in `keyword.py` gives a list of keywords, which cannot be used as variable names.

```
import keyword

print "Python keywords: ", keyword.kwlist
```

Keyword.py

The result of running this code is given by:

```
Python keywords:  ['and', 'as', 'assert', 'break', 'class', '
    continue', 'def', 'del', 'elif', 'else', 'except', 'exec', '
    finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is'
    , 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return',
    'try', 'while', 'with', 'yield']
```

Similarly, to check if a particular variable name is a keyword or not, one can use the function `keyword.iskeyword()`. As shown below, if the input string is a keyword then the value `True` is returned, otherwise `False` is returned.

```
>>>import keyword
>>>keyword.iskeyword('lambda')
True
>>>keyword.iskeyword('lamb')
False
```

4.3 Assignment Operator

The concept of variables used a symbol `=` which is not same as "equal to" in mathematics. Instead its one of the assignment operator. Below is the list of assignment operators.

Operator	Example
`=`	`v = a+b`
`+=`	`v +=a` ⇒ `v = v + a`
`-=`	`v -=a` ⇒ `v = v - a`
`/=`	`v /=a` ⇒ `v = v / a`
`//=`	`v //=a` ⇒ `v = v // a`
`*=`	`v *=a` ⇒ `v = v * a`
`**=`	`v **=a` ⇒ `v = v ** a`
`%=`	`v %=a` ⇒ `v = v % a`

Assignment operators are most frequently used feature on all kinds of programs. Increment and decrement operators like `+=` and `-=` respectively, are used extensively where we need to proceed stepwise.

Multiple assignment within the same statement can be done using `=` operator as follows:

```
a = b = c = 10
a
10
b
10
c
10
```

While assigning a value, its data type need not be explicitly defined. It is judged by python interpreter by the data itself. i.e. 4.0 will be taken as floating point number, 4 will be taken as integer, A single character 'a' or a group of characters like 'sandeep' will be taken as a string. This is shows in the following example of code.

```
a = 4.0; type(a)
float

a = 4; type(a)
```

```
int

a = 4e1;type(a)
float

b = 'a'; type(a)
int

b = 'a'; type(b)
str

b = 'sandeep'; type(b)
str
```

`4e1` denotes the **engineering notation** for number 4×10^1.

4.4 Arithmetic operators

Mathematical operators like `+,-,%,/` work by the same logic as in mathematics. a^b is written as `a**b` and floor division is given by `//` symbols.

```
>>>4.2 % 2.3
1.9000000000000004

>>>4.2 / 2.3
1.8260869565217392

>>>4.2 + 2.3
6.5

>>>4.2 - 2.3
1.9000000000000004

>>>4.2 ** 2
17.64

>>>4.2 // 2.3
1.0
```

Some Arithmetic operators work on *string* and *list* object as well.

```
>>> 'hello' + ' ' + 'world' + '!'
'hello world!'
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]
>>> 'hello'*3
'hellohellohello'
>>> [1,2,3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 'hello'/3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
>>> [1,2,3] // 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for //: 'list' and 'int'
>>> 'hello' % 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
>>> [1,2,3] % 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'list' and 'int'
>>> [1,2,3] - 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>> 'hello' - 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Using `+` and `*` operator on strings and list produce the effect of concatenation. Whereas `+` concatenates the two or more strings or list objects it is operated upon, `*` concatenates them `m` number of times where `m` is an integer used after the operators. The behaviour of `*` on strings and lists can be understood if one considers mathmatical multiplication in terms of addition i.e. $2*5 = 2+2+2+2+5$. Hence multiplication with `m` means adding `m` times.

4.5 Changing and defining data type

Data types of objects can be changed as per their definitions.

```
>>>int(4.2345)
4

>>>int(4.7345)
4

>>>float(4)
4.0

>>>float('sandeep')
---------------------------------------------------------------------------
ValueError                Traceback (most recent call last)
<ipython-input-26-ebab53faf0bc> in <module>()
----> 1 float('sandeep')

ValueError: could not convert string to float: sandeep
```

4.6 Order of usage

Python follows **PEMDAS** (Parenthesis Exponents Multiplication Division Addition Subtraction) order of operation. Hence, during a complex calculation involving a number of arithmetic operators, entities are calculated in the order : Parenthesis → Exponents → Multiplication → Division → Addition → Subtraction.

```
>>> 5 + (6 - 5) * 10 / (-1 / 9)
-5
>>> 5 * 5 + 5 - 4 ** 2
14
```

4.7 Logical operators

Logical operators are supremely important for comparing the objects. Operators used for comparison are called logical operators. Following is a table of python's logical operators:

Operator Symbol	**Operator meaning**	**Example**
`==`	equal to	`1==1` is `True`, `1==2` is `False`
`!=`	not equal to	`1!=1` is `False`, `1==2` is `True`
`<>`	not equal to	`1==1` is `False`, `1==2` is `True`
`<`	less than	`1<2` is `True`, `2<1` is `False`
`>`	greater than	`1>2` is `False`, `2>1` is `True`
`<=`	less than equal to	`1<=1` is `True`, `1<=2` is `True`
`>=`	greater than equal to	`1>=1` is `True`, `1>=2` is `False`

The result of logical operators is either of the two binary objects aptly named `True` and `False`. In some programming languages, binary operators are represented as `1` and `0`. They can also be compared for equality.

```
>>> not True
False
>>> a = True
>>> b = False
>>> a and b
False
>>> a or b
True
>>>
>>1 >= 2 == 2 >= 1
False
>>1 >= 2
False
>>>2 >= 1
True
>>>False == True
False
>>>False > True
False
>>>False < True
True
```

4.8 Membership Operator

Membership operator `in` checks if a value(s) of variables is a member of specified sequence. If the member is found, it returns the boolean value `True`, otherwise it returns `False`.

```
>>> 'hello' in 'hello world'
```

```
True
>>> 'name' in 'hello world'
False
>>> a=3
>>> b=[1,2,3,4,5]
>>> a in b
True
>>> 10 in b
False
```

Operator **in** is used extensively in checking conditions for loops. It is one of the most convenient way to run a loop. One constructs a list of as per a defined condition/formula and then runs a loop untill the condition is satisfied. This approach will become smore clear on chapter for functions and loops.

4.9 Identity Operator

To check if two values points to same type of object, an identify operator `is` is used. It returns a boolean value `True` if objects on its either side are same and return `False` otherwise.

```
1 >>> 1 is 1
2 True
3 >>> 1 is 1.0
4 False
5 >>> 1 is 2
6 True
```

At line 1, both objects i.e. `1` are `int` type whereas at line 3, left hand side has `int` and right hand side has `float`. Hence the result for `1 is 1.0` is given by boolean value `False`. At line 5, both `1` and `2` are `int`, hence the result is true again.

`is not` operator is negation of result with `is` operator.

```
>>> 1 is not 1.0
True
>>> 1 is not 1
```

```
False
```

4.10 Bitwise operators

All data is stored as bits in computers. If we can operate directly on bits, it will provide great flexibility and fast computation. But it is difficult to comprehend by humans since we are used to numerals defined in decimal format rather than binary format. A list of bitwise operators is presented below:

Bitwise Operator	**Description**
>>	Bitwise left shift
<<	Bitwise right shift
&	Bitwise AND
|	Bitwise OR
~	Bitwise not

- AND is 1 only if both of its inputs are 1, otherwise it's 0.
- OR is 1 if one or both of its inputs are 1, otherwise it's 0.
- XOR is 1 only if exactly one of its inputs are 1, otherwise it's 0.
- NOT is 1 only if its input is 0, otherwise it's 0.

Truth tables are useful in understanding thier operations.

AND	0	1
0	0	0
1	0	1
OR	0	1
0	0	1
1	1	1
XOR	0	1
0	0	1
1	1	0
NOT	0	1
	1	0

The use of these operators is mentioned below

```
>>> print bin(1), oct(1), hex(1)
0b1 01 0x1
>>> bin(10)
'0b1010'
>>> bin(1)
'0b1'
>>> 0b1010 >> 0b1
5
>>> bin(5)
'0b101'
# 0b1010 transforms into 0b101 by shifting one bits to right
>>> bin(2)
'0b10'
>>> 0b1010 >> 0b10
2
>>> bin(2)
'0b10'
# 0b1010 transforms into 0b10 by shifting two bits to right
>>> 0b1010 << 0b1
20
>>> bin(20)
'0b10100'
>>> 0b1010 << 0b10
40
>>> bin(40)
'0b101000'
# here left shifting is done by adding 0s to right
>>> 0b1010 & 0b10
2
>>> bin(2)
'0b10'
# AND (&) is 1 only if both of its inputs are 1, otherwise it's
    0
# The zero bits in this case effectively act as a filter,
    forcing the bits in the result to be zero as well
>>> 0b1010 | 0b10
10
>>> bin(10)
'0b1010'
OR is 1 if one or both of its inputs are 1, otherwise it's 0
```

4.10.1 Using bitwise operations

Bitwise operations find their use while dealing with hardware registers in embedded systems. Every processor uses one or more registers (usually a specific memory address) that control whether an interrupt is enabled or disabled. When an interrupt is enabled, signals can be communicated. Interrupts are enabled by setting the enable bit for that particular interrupt and most importantly, not modifying any of the other bits in the register. When an interrupt communicates with a data stream, it typically sets a bit in a status register so that a single service routine can determine the precise reason for the interrupt. Testing the individual bits allows for a fast decode of the interrupt source. This is where bit operations comes handy. Shift operators are used to shift the bits as per a formula whereas AND and OR operations are used to check the status of bits at a specific location. The same concept is used to alter the system file permission. In Linux file system, each file has a number called its *mode*, which indicates the permission about accessing the file. This integer can be retrieved in a program to know the status of permissions for the file. Example: `if ((mode & 128) != 0) {<do this>}` will check the mode by checking if an appropriate bit is 0, in 128 bit-system. Bitwise operations are also preferred for their speed of operation since they directly operate on bits in the memory.

4.11 Summary

Operators plays a very important part in computing as they provide the backbone of defining pathways for computing. All mathematical functions are expressed wither by individual operators or by combination of them. For a programming language that caters to a variety of fields like science, engineering, business, arts etc, a lot of different kinds of operators are needed. Python is now being applied in various dimensions of life and maturing with rich library of in-built as well as module wise operators.

5 Arrays

5.1 Introduction

Most often during scientific computation, a series of numbers needs to be operated upon together. The `list` data type stores a number of values within the same variable name. All elements of `list` can be accessed by their index. But individual `list` elements can belong to any data type. Hence a new kind of object needs to be defined, similar to list, but which stores only numeric values. This data type is called an array.

The `numpy` modules carries a unique object class called `array`. It carries only one data type as per its initial definition in the program. The concept of using arrays to store numerals, gave rise to a powerful idea of array based computing. The origins of this method can be traced back to matrix algebra. A matrix is also a collection of numbers. Similar to matrices, arrays can be multi dimensional and can be used to be operated upon by operators defined same as that for mathematical matrices. Using matrices, problems involving a system of equations can be solved i.e. solving many equations (which can even be coupled to each other) in one instance. Using the method of indexing of elements, particular elements can be accessed for operations. Using the concept of slicing, array dimensions can be altered as per requirements. Using operators acting on this object, mathematical formulations can be implemented. Present chapter will discuss the use of arrays for mathematical computations.

5.2 Numpy

The `numpy` package contains various items which can be used for numerical computation, hence the name *numerical python*. NumPy originated from *Numeric* which was originally created by Jim Hugunin along with contributions from several other developers. Travis Oliphant created NumPy in 2005, by incorporating features of the competing *Numarray* into Numeric, with extensive modifications. `numpy` is released under open source license. Present chapter has been tested for version 1.8.2 (stable at 1 March 2015).

`numpy` can be installed on Ubuntu 14.04 by a simple *pip* program as

```
pip install numpy
```

To use, it can be imported and version number can be checked by:

```
1 >>> import numpy
2 >>> print numpy.version.version
3 1.8.2
```

Line 1, import whole module named `numpy` for our use. Line 2 uses a function `version` which further uses further a function `version` to find out the installed version of `numpy` on the system. Users are encouraged to check their version of python.

Python packages are installed at `/usr/lib/python2.7/dist-packages/`. Here one can find `numpy`. To check the contents of the package, one can issue UNIX shell commands as follows:

```
$cd /usr/lib/python2.7/dist-packages/numpy
$ls
add_newdocs.py    __config__.pyc   distutils    fft
    __init__.pyc  matlib.py   oldnumeric   setup.pyc    version.pyc
```

```
add_newdocs.pyc   core             dual.py     _import_tools.py
    lib             matlib.pyc   polynomial   testing
compat            ctypeslib.py     dual.pyc    _import_tools.pyc
    linalg          matrixlib    random       tests
__config__.py     ctypeslib.pyc    f2py        __init__.py
    ma              numarray     setup.py     version.py
```

Now by issuing UNIX shell command `cd numarray`, one can see how python programs are defined to work with arrays. Explanation of these programs is beyond the scope of this introductory text on python, but the digital adventurer would like to explore them on a text editor to understand how modules work and in particular, how `arrays` in `numpy` works.

5.3 ndarray

`ndarray` is the main object of numpy, which is homogeneous multidimensional array. It is termed homogeneous since it can contain only one data type. Also it can be multidimensional as seen in examples below. They are indexed by a tuple of positive integers.

```
>>>a = [1,2,3]
>>>a
[1, 2, 3]
>>>type(a)
list
>>>import numpy
>>>b = numpy.array([1,2,3])
>>>b
array([1, 2, 3])
>>>type(b)
numpy.ndarray
>>>b.dtype
dtype('int32')
>>>c = numpy.array([1.0, 2.0, 3.0])
>>>c
array([ 1.,  2.,  3.])
>>>c.dtype
dtype('float64')
```

Above example explains how `list` and `arrays` in `numpy` are created differently. The `numpy` object `array` takes a list as input. Line 12 explains that element of array `b` are of the type `int32` i.e. 32 bit integers. Similarly, `c` is defined to have floating point numbers. The data type can be defined at the time of creation too.

```
>>>a1 = numpy.array([1,2,3], dtype=float)
>>>a1
array([ 1.,  2.,  3.])
>>>a1.dtype
dtype('float64')
>>>a2 = numpy.array([1,2,3], dtype=complex)
>>>a2
array([ 1.+0.j,  2.+0.j,  3.+0.j])
>>>a2.dtype
dtype('complex128')
```

`ndarray` is also known by its alias `array`. Apart from knowing the data type using `dtype`, there are a variety of methods to get information about various attributes of `ndarray`

ndarray.dtype	Data type of elements
ndarray.ndim	Dimension of array
ndarray.shape	Shape of array, (n,m) for (n,m) array
ndarray.size	Size of array ,$n \times m$
ndarray.itemsize	size in bytes of each element
ndarray.data	Buffer data containing actual element
ndarray.reshape	reshapes keeping $n \times m$ constant

The above table can be understood using the code below. We define a 3 array named a3.

```
>>>a3 = numpy.array( [ (1,2,3), (4,5,6), (2,7,8) ] )
>>>a3
array([[1, 2, 3],
       [4, 5, 6],
       [2, 7, 8]])
>>>a3.ndim
2
>>>a3.size
9
>>>a3.shape
```

```
(3L, 3L)
>>>a3.dtype
dtype('int32')
>>>a3.itemsize
4
>>>a3.data
<read-write buffer for 0x00000000085BAFE0, size 36, offset 0 at
    0x0000000009A1D2D0>
>>>a3.reshape(1,9)
array([[1, 2, 3, 4, 5, 6, 2, 7, 8]])
# reshapes a 3 X 3 array to 1 X 9 array (1 row and 9 coloumns)
>>>a3.reshape(9,1)
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [2],
       [7],
       [8]])
# reshapes a 3 X 3 array to a 9 X 1 array (9 rows and 1 coloumns
    )
>>>a3.reshape(9,1) is a3.reshape(1,9)
False
# result is false because both arrays have different shapes
```

5.4 Automatic creation of arrays

Various functions exists to automatically create an array of desired dimensions and shape. This comes handy during big mathematical calculations where creating arrays by hand is tiresome task.

5.4.1 zeros

To create an array where all elements are 0, we use `zeros()` function.

```
>>>zeros( (3,4) , dtype=float)

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

During initialization to zero values for matrix computations, `zeros()` function is extensively used.

5.4.2 ones

To create an array where all elements are 1, we use `ones()` function.

```
>>>ones( (3,4) , dtype=float)

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

5.4.3 ones like

Taking cue from an existing array, `ones_like()` creates an `ones` array of similar shape and type.

```
>>>a = np.array([[1.1, 2.2, 4.1],[2.5,5.2,6.4]])
>>>a

array([[ 1.1,  2.2,  4.1],
       [ 2.5,  5.2,  6.4]])

>>>ones_like(a)

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

5.4.4 empty

`empty()` returns a new array of given shape and type, without initializing entries.

```
>>>empty( (2,2) )

array([[   1.85323233e-316,    1.48523169e-316],
       [   1.48523169e-316,    3.15208230e-316]])
```

5.4.5 empty like

Taking cue from an existing array, `empty_like()` creates an `empty` array of similar shape and type.

```
>>>a = np.array([[1.1, 2.2, 4.1],[2.5,5.2,6.4]])
>>>a

 array([[ 1.1,  2.2,  4.1],
       [ 2.5,  5.2,  6.4]])

>>>empty_like(a)

 array([[  4.54892823e+174,   1.77289997e+160,   6.56350603e
       -091],
       [  7.67547114e-042,   4.57749997e-315,   2.47032823e-323]])
```

5.4.6 eye

Similar to a identity matrix, `eye()` returns a two dimensional array where diagonal elements are 1.

```
>>>eye(3, k=0)

 array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

# k is index of diaginal
# k = 0 means diagonal in center
# k = positive integer means it is shifted in upper triangle
# k = negative interger means it is shifted in lower traingle

>>>eye(3, k=1)

 array([[ 0.,  1.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  0.,  0.]])

>>>eye(3, k=2)

 array([[ 0.,  0.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>>eye(3, k=-1)
```

```

array([[ 0.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])

>>>eye(3, k=-2)

array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

5.4.7 identity

`identity()` function generates a two dimensional identity array.

```
identity(4)
Out[67]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

5.4.8 full

`full` fills up particular data into all elemental positions.

```
>>>full((3, 2), 5)

array([[ 5.,  5.],
       [ 5.,  5.],
       [ 5.,  5.]])
```

5.4.9 full like

Just like `empty_like` and `ones_like`, `full_like` creates a new matrix taking shape and data types from an existing array.

```
>>>a = np.array([[1.1, 2.2, 4.1],[2.5,5.2,6.4]])

array([[ 1.1,  2.2,  4.1],
      [ 2.5,  5.2,  6.4]])

>>>full_like(a,5)

array([[ 5.,  5.,  5.],
      [ 5.,  5.,  5.]])
```

5.4.10 random

To create a random array (filled up with random numbers), one uses the random function as follows:

```
a1 = random.rand(4)

a1
Out[89]: array([ 0.91994147,  0.75093653,  0.03770014,
    0.82726801])

a2 = random.rand(4,4)

a2
Out[91]:
array([[ 0.04817544,  0.96832776,  0.94496133,  0.13974019],
[ 0.88772227,  0.55457598,  0.54588295,  0.8659888 ],
[ 0.98772077,  0.93785153,  0.32630535,  0.20258845],
[ 0.28838472,  0.90353493,  0.50091164,  0.76243246]])
```

Note that the function `rand()` comes inside the subpackage `random`. To get complete details of this wonderful package, user is encouraged to explore it using `help(numpy.random)` after issuing the command `import numpy`.

This gives the following description of the package:

```
>>> help(numpy.random)
DESCRIPTION
========================
Random Number Generation
========================

=====================
Utility functions
=====================
random               Uniformly distributed values of a given
    shape.
bytes                Uniformly distributed random bytes.
random_integers      Uniformly distributed integers in a given
    range.
random_sample        Uniformly distributed floats in a given
    range.
permutation          Randomly permute a sequence / generate a
    random sequence.
shuffle              Randomly permute a sequence in place.
seed                 Seed the random number generator.
====================

====================
Compatibility functions
====================
rand                 Uniformly distributed values.
randn                Normally distributed values.
ranf                 Uniformly distributed floating point
    numbers.
randint              Uniformly distributed integers in a given
    range.
====================

====================
Univariate distributions
====================
beta                 Beta distribution over ``[0, 1]``.
binomial             Binomial distribution.
chisquare            :math:`\chi^2` distribution.
exponential          Exponential distribution.
f                    F (Fisher-Snedecor) distribution.
gamma                Gamma distribution.
geometric            Geometric distribution.
gumbel               Gumbel distribution.
hypergeometric       Hypergeometric distribution.
laplace              Laplace distribution.
logistic             Logistic distribution.
lognormal            Log-normal distribution.
logseries            Logarithmic series distribution.
negative_binomial    Negative binomial distribution.
noncentral_chisquare Non-central chi-square distribution.
noncentral_f         Non-central F distribution.
normal               Normal / Gaussian distribution.
pareto               Pareto distribution.
poisson              Poisson distribution.
power                Power distribution.
rayleigh             Rayleigh distribution.
triangular           Triangular distribution.
```

```
uniform               Uniform distribution.
vonmises              Von Mises circular distribution.
wald                  Wald (inverse Gaussian) distribution.
weibull               Weibull distribution.
zipf                  Zipf distribution over ranked data.
======================

======================
Multivariate distributions
======================
dirichlet             Multivariate generalization of Beta
    distribution.
multinomial           Multivariate generalization of the binomial
     distribution.
multivariate_normal   Multivariate generalization of the normal
    distribution.
======================

======================
Standard distributions
======================
standard_cauchy       Standard Cauchy-Lorentz distribution.
standard_exponential  Standard exponential distribution.
standard_gamma        Standard Gamma distribution.
standard_normal       Standard normal distribution.
standard_t            Standard Student t-distribution.
======================

======================
Internal functions
======================
get_state             Get tuple representing internal state of
    generator.
set_state             Set state of generator.
======================
```

The above description shows a rich library of functions to create random numbers as per choice. This makes `numpy` a good choice for libraries used in simulation work.

5.4.11 diagonal

`diag()` commands makes an array of defined dimensions as follows:

```
>>>a1 = random.randn(4,4)
```

```
>>>a1

array([[ 0.32300659, -0.80867401,  0.73055204, -0.42193636],
[ 0.26766307, -1.41864706, -0.52676398, -1.68007247],
[-0.39765223,  0.40380447,  0.51565046,  1.18807724],
[ 1.01937589,  1.58661357, -0.86241172, -0.86339454]])

>>>a2 = diag(a1,k=0)
>>>a2
array([ 0.32300659, -1.41864706,  0.51565046, -0.86339454])
# an array of true diagonal elements of array 'a2' is returned.

>>>a2 = diag(a1,k=1)
>>>a2
array([-0.80867401, -0.52676398,  1.18807724])
# an array of diagonal elements of array 'a2' is returned where
    dimesnional axis is shifted upwards by one unit.

>>>a2 = diag(a1,k=-2)
>>>a2
array([-0.39765223,  1.58661357])
# an array of diagonal elements of array 'a2' is returned where
    dimesnional axis is shifted downwards by two unit.
```

5.5 Numerical ranges

Creating a sequence of numbers is an integral part of a numerical computation. A variety of functions exists to create a sequence of numbers automatically.

5.5.1 arange

The syntax for automatically generating a range of numbers from a starting point to a stop point with a step size is given by

`numpy.arange([start, ]stop, [step, ]dtype=)`

```
>>>arange(1,10,0.5)

array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
    5.5,  6. ,
6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])

# default value of start is 1
```

```
>>>arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

#default value of step size is 1
>>>arange(1,10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

5.5.2 linspace

Whereas `arange()` has good control over step size, one cannot specify number of elements in the array. To solve this issue `linspace()` function is defined with following syntax.

`linspace(start, stop, num, endpoint, dtype)`

```
>>>linspace(1,10,5)
array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])

>>>linspace(1,10,5, endpoint=False)
array([ 1. ,  2.8,  4.6,  6.4,  8.2])
```

5.5.3 logspace

Just as linearly spaced points are generated by `linspace()`, `logspace()` generates linearly spaced points on a logarithmic axis.

`logspace(start, stop, num, endpoint=, base=, dtype=)`

```
>>>logspace(2.0, 5.0, num=3)
array([    100.        ,    3162.27766017,  100000.        ])

>>>logspace(2.0, 5.0, num=3, base=2)
array([  4.        ,  11.3137085,  32.        ])

>>>logspace(2.0, 5.0, num=3, endpoint=False)
array([   100.,    1000.,   10000.])
```

5.5.4 meshgrid

The `meshgrid` is modeled after MATLAB ®`meshgrid` command. To understand the working of `meshgrid`, its best to use it once as follows.

```
>>>xx = linspace(1,3,3)
>>>xx
array([ 1.,  2.,  3.])
>>>yy = linspace(2,4,3)
>>>yy
array([ 2.,  3.,  4.])

>>>(a,b) = meshgrid(xx,yy)
>>>a

array([[ 1.,  2.,  3.],
      [ 1.,  2.,  3.],
      [ 1.,  2.,  3.]])

>>>b

array([[ 2.,  2.,  2.],
      [ 3.,  3.,  3.],
      [ 4.,  4.,  4.]])

# another example

>>>x = numpy.array([1, 2, 3])
>>>y = numpy.array([10, 20, 30])
>>>XX, YY = numpy.meshgrid(x, y)
>>>ZZ = XX + YY

>>>ZZ
array([[11, 12, 13],
      [21, 22, 23],
      [31, 32, 33]])
```

`meshgrid()` makes a two dimensional coordinate system where x-axis is given by first argument (here `xx`) and y-axis is given by second argument (here `yy`). This function is used while plotting three dimensional plots or defining a function defined on two variables.

5.5.5 mgrid and ogrid

`mgrid` and `ogrid` are used to created mesh directly i.e. without using `linspace, arange` etc. A simple statement like

`mgrid[a:b , c:d]`

constructs a grid where x-axis has points from `a` to `b` and y-axis has points from `c` to `d`.

`mgrid` constructs a multidimensional `meshgrid`. Following example explains its use.

```
>>>(a,b) = mgrid[1:10 , 2:5]
>>>a

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

>>>b

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

>>>x, y = np.ogrid[0:5, 0:5]
>>>x

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

```

```
>>>y

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

5.5.6 tile

`tile()` functions makes the copy of existing array by the defined number of times to make a new array as follows:

```
>>>a = array([1,2,3])
>>>b = tile(a,3)
>>>b

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

# array a is repeated 3 times to make a new array b

# Another example to do the same for two dimesnional array

>>>a1 = eye(4)
>>>a1

array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

>>>a2 = tile(a1,2)
>>>a2

array([[ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.]])
>>>a2 = tile(a1,(2,2))
>>>a2

array([[ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.],
       [ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  1.]])

```

```
# here the extension is done on both dimensions
```

5.6 Broadcasting

Basic operations on `numpy` arrays are element-wise. This needs that dimensions of the arrays should be compatible for the desired operation i.e. 2 arrays and 2 arrays would be incompatible as first array has one less column than the second one.

```
>>>a = eye(4)
>>>b = array([1,2,3,4])
>>>c = a + b
>>>c

array([[ 2.,  2.,  3.,  4.],
      [ 1.,  3.,  3.,  4.],
      [ 1.,  2.,  4.,  4.],
      [ 1.,  2.,  3.,  5.]])

>>>a.shape
(4L, 4L)

>>>b.shape
(4L,)

>>>c.shape
(4L, 4L)

# bradcasting enables array a ( 4 X 4) to be added to b (4 X 1)
    to produce an array c (4 X 4)

# ANother example
>>>a = eye(4)
>>>a

array([[ 1.,  0.,  0.,  0.],
      [ 0.,  1.,  0.,  0.],
      [ 0.,  0.,  1.,  0.],
      [ 0.,  0.,  0.,  1.]])

>>>b = array([10, 10, 10, 10])
>>>c = a + b
>>>c

array([[ 11.,  10.,  10.,  10.],
    [ 10.,  11.,  10.,  10.],
    [ 10.,  10.,  11.,  10.],
    [ 10.,  10.,  10.,  11.]])

# A 4 X 4 matrix can be operated with a 4 X 1 matrix by making
    the 'invisible' elemets zero.
```

5.7 Indexing

Elements of a array or list start with 0 in python i.e. first element is indexed 0. All elements can be accessed using their indexes.

```
>>>a =[1,2,3,4,5,6]
>>>type(a)
 list # a stores a 'list' object
>>>b = array(a)
>>>type(b)
numpy.ndarray # b stores a 'array' object
>>>a[1]
2 # accessing second element from left hand side for the list 'a
    '
>>>b[0]
1 # accessing first element from left hand side for the array 'b
    '
>>>a[-1]
6 # accessing the first element from the right hand side for
    list 'a'
>>>b[-2]
5 # accessing the second element from the right hand side for
    array 'b'
```

Above examples make it clear that arrays are simply homogeneous lists and follow the same rules of indexing. Multidimensional arrays also follow the same pattern of indexing. For two dimensional arrays, first number indicates the row and second number indicates the columns.

```
>>>a1 = array([[1,2,3],[3,2,1]])
>>>a1

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

```

```
>>>a1[1,2]
1 # choosing elemnet whose row is indexed 1 and coloumn is
    indexed 2 i.e second row and thierd coloumn i.e down-right
    last element

>>>a1[1,1]
2 # chooisng an element whose row is indexed 1 and clolumn is
    indexed 1 i.e. second row and second coloumn

>>>a1[1]
array([3, 2, 1]) # chooisng row with index 1 i.e second row
```

Indexes can be used to assign a particular value of the element too.

```
>>>a1 = array([[1,2,3],[3,2,1]])
>>>a1

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

>>>a1[1,1] = 0
>>>a1

array([[1, 2, 3],
     [3, 0, 1]])
# second row and second coloumns element i.e 2 is changed to 0
```

5.8 Slicing

Amongst the first operations to be applied on arrays is **slicing**. Slicing employs the operator **:** which is used to separate the data on the row.

```
>>>a1 = arange(10)
>>>a1
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>>a1[0:5]
array([0, 1, 2, 3, 4])
# [0:5] selecets 'from' index 0 'untill' index 5 i.e. excluding
    index 5

>>>a1[:5]
```

```
array([0, 1, 2, 3, 4])
# [:5] selecets 'from' starting 'untill' index 5 i.e. excluding
    index 5

>>>a1[2:5]
array([2, 3, 4])
# [2:5] selecets 'from' index 2 'untill' index 5 i.e. excluding
    index 5

>>>a1[2:-2]
array([2, 3, 4, 5, 6, 7])
# [2:-2] selecets 'from' index 2 'untill' index -2 (counting
    from right starts from -1) i.e. exclusing index -2

>>>a1[2:]
array([2, 3, 4, 5, 6, 7, 8, 9])
# [2:] selecets 'from' index 2 'untill' last index

>>>a1[0:5:2]
array([0, 2, 4])
# [start:stop:step] =[0:5:2] hence it takes a tep of 2 while
    choosing indices from 0 to 5
```

slicing can also be accomplished for multidimensional arrays.

```
>>>a = [1,2,3,4,5]
>>>b = [5,6,7,8,9]
a1 = array([a,b])
>>>a1

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

>>>a1.ndim
2 # the dimension of array a1 is 2

>>>a1[0:2,0:2]

array([[1, 2],
     [5, 6]])
# Start collecting elements from row indexed 0 and coloumn
    indexed 2

>>>a1[0:2,0:4]

array([[1, 2, 3, 4],
    [5, 6, 7, 8]])
# First slice indicates to collect only first two elements by
    second slice indicates to collect first four elements, hence
    using broadcasting the result is implemented

>>>a1[1:2,-1:]
array([[9]]) # because second slice indicates to collect the
    last element
```

Similar logic can be applied to any dimensional array. Slicing becomes an extremely important tool for data filtering. In some cases, we would like to work with only specific rows and/or columns of data. In that case, the data can be sliced as per need.

5.9 Copies and views

From the memory usage point of view, slicing operation creates just a **view** of original array. Using `numpy.may_share_memory()`, one can verify this claim.

```
>>>a = arange(10)
>>>a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>b = a[2:5]
>>>b
array([2, 3, 4])
>>>may_share_memory(a,b)
True
# array 'a' and 'b' share same memory space

# Now if we change first element of 'b', array 'a' element will
    also change since 'b' is just a "view" of 'a'
>>>b[0] = 10
>>>a
>>>array([ 0,  1, 10,  3,  4,  5,  6,  7,  8,  9])
```

Being only a view, if an element of a slice is modified, original array is modified too. Whereas this facilty can be desirable, it can create nuisance in certain problems. Hence function `copy()` provides a way out by copying original array instead of providing a view.

```
>>>a = arange(10)
>>>c = a[2:5].copy()
>>>c # checking elements of array 'c'
array([2, 3, 4])
>>>c[0] = 10 # changing first element to 10
>>>c # checking for change
array([10,  3,  4])
>>>a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# array 'a' remains unchanged
>>>may_share_memory(a,c)
False
# 'a' remains unchanged because 'a' and 'c' don't share thier
    memories
```

5.10 Masking

Arrays can be indexed using the method of masking. Masking is a way to define the indexes as a separate object and then generating a new array from original array using the mask as a rule. There are two ways using which arrays can be masked: fancy indexing and indexing using boolean values. It is important to note that masking methods generate **copies** instead of views. The two methods are discussed in the following subsections.

5.10.1 Fancy indexing

`numpy` offers quite unique indexing facilities. One of them is fancy indexing where an array of indices can be used to generate an array of elements.

```
>>>a = arange(1000)**3
# Generated cubes of first 1000 cubes
>>>i = array(arange(10))
# Generated an array of first 10 numbers starting from 0 upto 9
>>>a[i]
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
# Above is the array of first 10 cubes
>>>j = array(arange(0,50,10))
>>>j
array([ 0, 10, 20, 30, 40])
# j is an array of numbers from 0 to 50 with steps of 10
>>>a[j]
```

```
array([     0,   1000,   8000, 27000, 64000])
# a[j] is the array of cubes indexed with array j
>>>k = array( [ [ 1, 2], [ 11, 12 ] ] )
# k is a two dimensional array of indexes 1,2,11,12
>>>a[k]
array([[    1,     8],
       [1331, 1728]])
# a[k] is array made up of elements placed at indexes given by k
```

5.10.2 Indexing with Boolean arrays

Indexing using integers specify the position of the element and using fancy indexing, one can pick up those particular elements. Using boolean data type for indexing, this is done with a different philosophy. Here boolean value `True` means that array should become part of final array and value `False` indicates that the element should not become part of the array.

```
>>>a = arange(100).reshape(10,10)
# a is a 10 X 10 matrix of first hundred numbers

# Our aim is to make an array of even numbers and make a 5 X 10
    matrix

>>>b = (a % 2 ==0)

b now stores a matrix of boolean values where the value is 'True
    ' when the element of a is divisible by two and value is '
    False' when elemnet of a is not divisible by two.

>>>a[b].reshape(5,10)

array([[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18],
     [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
     [40, 42, 44, 46, 48, 50, 52, 54, 56, 58],
     [60, 62, 64, 66, 68, 70, 72, 74, 76, 78],
     [80, 82, 84, 86, 88, 90, 92, 94, 96, 98]])
```

5.11 Arrays are not matrices

Even though the python object ndarray (or simply arrays) look like defining matrices, they are not the same. Matrices are defined using the object `numpy.matrix`. A `numpy.matrix`is a specialized 2-D array that retains its 2-D nature through operations. Certain special operators, such as `*` (matrix multiplication) and `**` (matrix power) are defined for them.

```
>>>a = arange(10).reshape(2,5)
# defining a 2 X 5 array made of numbers from first ten numbers
>>>a
array([[0, 1, 2, 3, 4],
     [5, 6, 7, 8, 9]])

>>>type(a)
numpy.ndarray
# 'a' is a numpy.ndarray

# Now we shall create a matrix using this array
>>>b = matrix(a)
>>>b

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

>>>type(b)
numpy.matrixlib.defmatrix.matrix
# 'b' is a matrix unlike 'a', which is an array

# Even though 'a' and 'b' looks similar, they are two different
    objects
```

Mathematical operations like scalar multiplication, matrix multiplication (dot and cross product), matrix power is defined for this data type.

```
>>>a_array = arange(12).reshape(3,4)
>>>a_array

array([[ 0,  1,  2,  3],
     [ 4,  5,  6,  7],
     [ 8,  9, 10, 11]])
# 'a_array' stores a 3 X 4 array of numbers

>>>a_array_1 = a_array.copy
# 'a_array_1' is a copy of 'a_array'

>>>sum_array = a_array + a_array
>>>sum_array

array([[ 0,  2,  4,  6],
     [ 8, 10, 12, 14],
     [16, 18, 20, 22]])
# sum of two arrays produces a new array where elementwise
    operation (addition here) is performed

>>>scalar_product = 3 * a_array
>>>scalar_product

array([[ 0,  3,  6,  9],
     [12, 15, 18, 21],
     [24, 27, 30, 33]])

# scalar product of array with a number is simply elementwise
    multiplication

>>>a_matrix = matrix (a_array)
>>>a_matrix

matrix([[ 0,  1,  2,  3],
     [ 4,  5,  6,  7],
     [ 8,  9, 10, 11]])
# A matrix 'a_matrix' is created using an array 'a_scalar'

>>>sum_matrix = a_matrix + a_matrix
>>>sum_matrix

matrix([[ 0,  2,  4,  6],
     [ 8, 10, 12, 14],
     [16, 18, 20, 22]])

>>>scalar_mul_matrix = 3 * a
>>>scalar_mul_matrix = 3 * a_matrix
>>>scalar_mul_matrix

matrix([[ 0,  3,  6,  9],
     [12, 15, 18, 21],
     [24, 27, 30, 33]])

# Checking for transpose
>>>a_array_T = a_array.T
>>>a_array_T

array([[ 0,  4,  8],
     [ 1,  5,  9],
     [ 2,  6, 10],
     [ 3,  7, 11]])

>>>a_matrix_T = a_matrix.T
>>>a_matrix_T
```

```

 matrix([[ 0,  4,  8],
      [ 1,  5,  9],
      [ 2,  6, 10],
      [ 3,  7, 11]])

# checking for dot product of arays and matrices

>>>dot_array = dot(a_array, a_array_T)
>>>dot_array

 array([[ 14,  38,  62],
      [ 38, 126, 214],
      [ 62, 214, 366]])

>>>dot_matrix = dot(a_matrix, a_matrix_T)
>>>dot_matrix

 matrix([[ 14,  38,  62],
      [ 38, 126, 214],
      [ 62, 214, 366]])


# Well the matrix behaves exactly same as array uptill now
```

But there are some differences too:

```
>>>power_array = a_array ** 2
>>>power_array

 array([[  0,   1,   4,   9],
      [ 16,  25,  36,  49],
      [ 64,  81, 100, 121]])

# But issuing a command a_matrix ** 2 will give an error because
     matrices can only be multiplied if number of rows of one is
    equal to number of coloumns of other i.e m X n can be
    multiplied with only n X 1
# Hence matrix power needs the matrix to be squared matrices
```

A common question that arises in the minds of programmers is that if one had `array` object then what was the need of `matrix` ?

The answer is quite complex. While `array` serves most of the general purposes for matrix algebra, `matrix` is written to facilitate linear algebra functionalities. Linear algebra is performed using a submodule of `numpy` accessed as `numpy.linalg`. Issuing command `help(numpy.linalg)` gives idea about the purpose of this module. Some of the useful function from matrix algebra point of view are:

- `solve()` to solve system of linear equations
- `norm()` to find norm of matrix
- `inv()` to find matrix inverse
- `pinv()` to find pseudo-inverse
- `matrix_power()` to perform an integer power of a square matrix

To perform linear algebra calculations using matrices, it is suggested that `matrix` object is used to avoid errors.

More information about matrix object can be found by issuing the command `help(numpy.matrix)` or visiting `http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html`.

5.12 Some basic operations

`array` allows some basic in-built operations which come quite handy while performing calculations. These functions have been written to optimize time spent on running the code and minimizing error, hence user can concentrate on using them for computation rather than writing their own and then optimizing them. Some of them have been discussed below.

5.12.1 sum

`sum()` calculates the sum of all elements in the array.

```
>>>a = arange(25)
# created an array 'a' consisting of first 25 numbers
>>>a

 array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
    14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24])

>>>sum(a)
300

# reshaping it to a 5 X 5 matrix does not change the sum of
    elements

>>>b = a.reshape(5,5)
>>>b

array([[ 0,  1,  2,  3,  4],
     [ 5,  6,  7,  8,  9],
     [10, 11, 12, 13, 14],
     [15, 16, 17, 18, 19],
     [20, 21, 22, 23, 24]])

>>>sum(b)
300

# sum can be defined for an axis

>>>sum(b, axis=0)
array([50, 55, 60, 65, 70])
# each element if sum of coloumn elements

>>>sum(b, axis=1)
array([ 10,  35,  60,  85, 110])
# each element is sum of row elements
```

5.12.2 Minimum and maximum

`min()` and `max()` gives the minimum and maximum value amongst the element values.

```
>>>a = arange(10).reshape(2,5)
>>>a

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

>>>a.min()
0
>>>a.max()
9
>>>a.max(axis=0)
```

```
array([5, 6, 7, 8, 9])
# maximum in each coloumn

>>>a.max(axis=1)
array([4, 9])
# maximum in each row
```

5.12.3 Statistics: mean median and standard deviation

`mean(), median(), std()` finds the mean median and standard deviation for the data stored in the array.

```
>>>a = arange(10).reshape(2,5)
>>>a
array([[0, 1, 2, 3, 4],
     [5, 6, 7, 8, 9]])
>>>mean(a)
4.5
>>>median(a)
4.5
>>>std(a)
2.8722813232690143
```

5.12.4 sort

`sort()` sorts the array values from maximum to minimum.

```
>>>a = rand(3,4)
>>>a

array([[ 0.12623497,  0.08767029,  0.76615535,  0.85825585],
     [ 0.78531643,  0.92799983,  0.03808058,  0.87323096],
     [ 0.40734359,  0.7030647 ,  0.02290688,  0.1080126 ]])

>>>sort(a)

array([[ 0.08767029,  0.12623497,  0.76615535,  0.85825585],
     [ 0.03808058,  0.78531643,  0.87323096,  0.92799983],
     [ 0.02290688,  0.1080126 ,  0.40734359,  0.7030647 ]])
# sorts by coloumn by default

>>>sort(a, axis=1)

```

```
array([[ 0.08767029,  0.12623497,  0.76615535,  0.85825585],
     [ 0.03808058,  0.78531643,  0.87323096,  0.92799983],
     [ 0.02290688,  0.1080126 ,  0.40734359,  0.7030647 ]])

>>>sort(a, axis=0)

array([[ 0.12623497,  0.08767029,  0.02290688,  0.1080126 ],
     [ 0.40734359,  0.7030647 ,  0.03808058,  0.85825585],
     [ 0.78531643,  0.92799983,  0.76615535,  0.87323096]])
```

A variety of sorting algorithms exist. Choice of algorithm can depend on requirements about average speed, worst case scenario, workspace size and stability. Numpy documentation at `http://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html` list three choices as follows:

kind	**speed**	**worst case**	**workspace**	**stable**
'quicksort'	1	$O(n^2)$	0	no
'mergesort'	2	$O(n \times log(n))$	$\frac{n}{2}$	yes
'heapsort'	3	$O(n \times log(n))$	0	no

Sorting an array of complex numbers is accomplished by `sort_complex()`.

```
>>>a = array([4-3j, 4+5j, 3-8j])
>>>sort_complex(a)
array([ 3.-8.j,  4.-3.j,  4.+5.j])
```

5.12.5 Rounding off

Rounding off numbers is performed by function `around()` with same logic as in mathematics, i.e. if a numeral is 5 or more, the preceding numeral is incremented by 1.

```
>>>a = rand(10)
>>>a

array([ 0.13817612,  0.05911436,  0.55986426,  0.10755959,
    0.62031418,
0.68802259,  0.40226421,  0.71521764,  0.34881375,  0.00660543])

>>>around(a)
array([ 0.,  0.,  1.,  0.,  1.,  1.,  0.,  1.,  0.,  0.])

```

```
>>>around(a).astype(int)
array([0, 0, 1, 0, 1, 1, 0, 1, 0, 0])
```

5.13 asarray and asmatrix

A variety of variables are not defined as arrays but if at certain point of time during computation, if they needs to be considered as an array or as matrix then `asarray()` and `asmatrix()` can be used.

```
>>>(a,b,c,d) = (1,2,3,4)
>>>array1 = asarray([a,b,c,d])
>>>array1
array([1, 2, 3, 4])
>>>matrix1 = asmatrix([a,b,c,d])
>>>matrix1
matrix([[1, 2, 3, 4]])
>>>string = 'Hello world'
>>>array2 = asarray(string)
>>>array2
array('Hello world', dtype='|S11')
# indicates that data type is strong with 11 characters
```

5.14 Summary

Array based computing is used as a primary force to solve equations or system of equations. Using slicing and indexing operations, it provides powerful tools to manipulate data using a program. Since present book is an interactive text in python, hence discussion about all functions for indexing and slicing is out of the scope for the book. Users are requested to visit `http://docs.scipy.org/doc/numpy/reference/routines.indexing.html`

Discussing all the facilities of array manipulations that are present in `numpy` is beyond the scope of any textbook. Moreover, new functionalities are added with each newer version. Some quite important functions ere discussed in the chapter. `http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#joining-arrays` discuss these array manipulations with ample of examples. Use is advised to consult the link as and when required or when in doubt about usage.

Python's ability to flexible create variety of arrays and compute using various mathematical functions makes it one of the most preferred language in the field of computational physics and mathematics. One more fact which makes it a preferable programming language among scientific community is its ability to plot publication quality graphs with relative ease. This shall be discussed in the next chapter.

6 Plotting

6.1 Introduction

Plotting the data is one of the most essential part of numerical computation. In per-processing, during computation and post-processing, plotting data in term of variety of graphs becomes essential. Visualization of data in a convenient format lets one understand the process better. Visual clues generate a lot of information about the process which generated that particular data. One can look for error easily and derive simple as well as complex interpretations. A good programming language must incorporate facilities to plot data easily. Plotting two dimensional (2D) and three-dimensional (3D) graphs are essential qualities in a good visualization product. Python users have a good number of choices in this regard.

Present chapter will discuss some of them. Essential requirements while choosing a plotting library, depends on requirements of data like:

- plotting 2D or 3D
- plotting live data or static

- plotting large data quickly
- saving plots in variety of formats
- plotting data with chosen resolution to keep a check about file size.

Various plotting libraries will be discussed in present chapter and they will be judged based on above mentioned parameters. None of them if perfect. They fulfill each others gaps. Hence a programmer is encouraged to learn all of them and then choose to use them as and when required.

6.2 Matplotlib

John Hunter, the creator of matplotlib, rightly quoted that

"Matplotlib tries to make easy things easy and hard things possible".

In some cases, with just one line of code, one can generate high quality publication ready graphics visualization of problem at hand. Before python, *gnuplot* was used to plot the data passed by a python script. With `matplotlib` at hand, this action has become very flexible now. `matplotlib` was modeled after graphic capabilities of MATLAB ®, which came as a boon for programmer who were already well versed with MATLAB ®. Some of the major advantages of using `matplotlib` over other plotting libraries are:

- It is integrated with LaTeXmarkup
- It is cross platform and portable
- Its is open sourced, so one does not have to worry about license fees.
- Being part of python, its programmable

`matplotlib` stands for mathematical plotting library. It is one of the most popular plotting library amongst programmer owing to its simple and intuitive commands and well as ability to produce high quality plots which can be saved in variety of formats. It supports both interactive and non-interactive modes of plotting and can save images in a variety of formats like (JPEG, PS, PDF, PNG etc). It can utilize a variety of window toolkits like GTK+, wxWidgets, QT etc. The most attractive feature is that it have a variety of plotting styles like line, scatter, bar charts as well as histograms and many more. It can also be used interactively with Ipython. `numpy` is essential to work with `matplotlib`, hence it must be installed on the system before one can work with `matplotlib`.

6.2.1 Build Dependencies

- **Python 2.x** (Python 3 is not supported yet)
- **numpy >1.1**
- **libpng >1.1**
- **FreeType >1.4**

6.2.2 pylab versus pyplot

Within `matplotlib`, `pyplot` and `pylab` are two most discussed modules inside `matplotlib` which provide almost same functionalities and thus cause some confusion about their usage. Hence its important to differentiate between them at this point.

Within the package `matplotlib`, two packages namely `matplotlib.pylab` and `matplotlib.pyplot` exists. Since plotting can start as a simple exercise and then become quite complicated one, `matplotlib` is designed in a hierarchical pattern where by default, simple functions are implemented in `matplotlib.pyplot` environment and then as the complexity increases, a more complex environment like `matplotlib.pylab` is implemented.

A more object oriented approach is `matplotlib.pyplot` where functions like `figures(), axes(), axes()` are defined as objects to keep track of their properties dynamically. For even complex tasks like making graphic user interfaces (GUI), exclusive `pyplot` usage can be dropped altogether and object-oriented approach can be used to fabricate plots. At basic level, when plots are mostly non-interactive, `pyplot` environment can be used. `pylab` is sued for interactive studies.

6.3 Plotting basic plots

We shall first explore the working environment offered by `matplotlib.pyplot`. Functional inside `pyplot` control a particular feature of the plot like putting up a tile, mentioning labels on x-axis and y-axis, putting mathematical equations on the body of plot at a desired position, defining tick labels, defining types of markers to plot a graph etc. `pyplot` is stateful i.e. it keeps updating the changes in the state of figure once defined. This makes it easier to modify a graph until desired level before including the code in the program.

`plot()` function is used for plotting simple 2D graphs. It take a number of arguments which fill data and other feature information to plot a graph. In its simplest form, it can plot a list of numbers (code: `sqPlot0.py`).

```
# plotting numbers
import numpy as np
import matplotlib.pyplot as plt
a = np.arange(10)
plt.plot(a)
plt.show()
```

sqPlot0.py

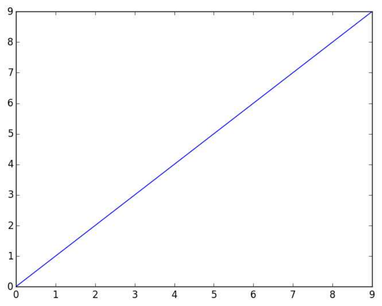


Figure 6.1: Plotting first ten integers using plot() function

The result can be seen in 6.1. A plot needs two axes which are usually termed as x and y axis. When `plot()` command is supplied with a single list or array, it assumes it to be the values for y-axis and automatically generates corresponding x-values taking cue from length of list. In or case, we had 10 numbers, hence x-axis had 10 numbers from 0 to 9.

`plot()` can take both axes as input vectors to produce a plot a shown in the code given by `sqPlot1.py`.

```
import numpy as np
from matplotlib import pylab as pl
x = np.linspace(0,100)
y = x ** 2
pl.plot(x,y)
pl.show()
```

sqPlot1.py

The result can be seen in figure **??**. A very basic plot could be plotted by just few lines of code where one first imports relevant libraries (line 1 and line 2), then define x and y axes, and then use the `plot` command which is given the parameters about x and y axes. These commands can e issued one by one at python command prompt, or it can be saved as a python file (use a text-editor, write the code and save with `sqPlot1.py`).

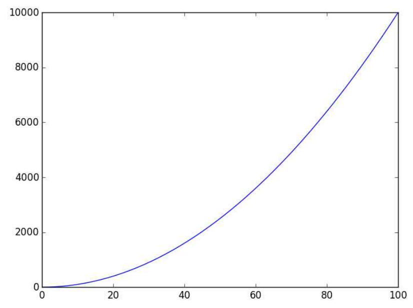


Figure 6.2: Plotting first ten integers using plot() function

The axes are defined as a `numpy` array. They can be generated by all the methods available at hand like generating by hand, generated using a formula (like in code `sqPlot1.py`, array named `y` is generated by element-wise squaring of array `x`), generated by a data file, data taken live from a remote/local server using internet or LAN etc. Subsequent chapters will deal with file input output facilities. Hence for present chapter, only array generated by self or using formulas, will be used.

It is worth mentioning the role of seemingly simple but powerful function `show()`. A simple search on command prompt (write: `matplotlib.pyplot.show`) for it declares its purpose. It displays figure(s) on a computer terminal having graphics capabilities, which most modern computers do. During non-interactive mode in `Ipython` console, it first displays all figures and block the console until the figures have been closed whereas in interactive mode it does not block the console. Both modes have thier own merits. Interactive mode is used for checking the change in features. Non-interactive mode is used when the focus is more upon the code generation which produces the graphs. Usually, programmers work with interactive mode and optimize a view of plots and then work with non-interactive mode taking the same settings which were generated during experiments with interactive mode.

6.3.1 Plotting more than one graph on same axes

More than one plot on same axes, can be plotted in same figure by simply issuing two plot commands as in `sqPlot2.py`

```
import numpy as np
from matplotlib import pylab as pl
x = np.linspace(0,100)
y1 = x ** 2 # y is square of x
y2 = x ** 2.2 # y is x raised to power 2.2
pl.plot(x,y1)
pl.plot(x,y2)
pl.show()
```

sqPlot2.py

6.3.2 Various features of a plot

A variety of features exists for a graph. Following is a list of features of a graph:

- **Title**: Title gives a short introduction for the purpose of the graph

  `title()` object sets the title of the current axes, positioned above axes and in the center. It takes a string as an input.

- **Labels for axes**: Labels marks the purpose of graph axes.

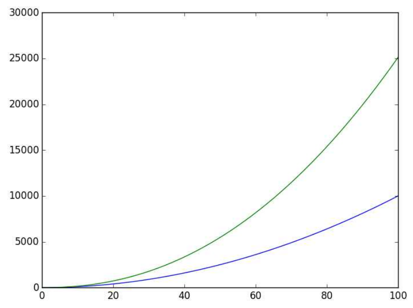


Figure 6.3: Plotting more than one graph on same axes

`xlabel()` and `ylabel()` object sets the label of x and y axis respectively. The text strong which it takes as input is positioned above the axis in the center.

- **Ticks**: Ticks on axis show the division of data points on an axes and help judging information about a data point on graph.

  `xticks` and `yticks` sets the ticking frequency and location. For example
  `xticks( arange(5), ('a', 'b', 'c', 'd', 'e') )`

  shows that 5 ticks named `a,b,c,d,e` are placed equidistant. `linspace` and `logspace` can also be used for the same.

- **Markers**: markers are the symbols drawn at each data point. The size and type of markers can be differentiated for showing the data points belonging to two or more different data sets.

  In the `plot()` function, for every pair x, y, there is an optional third argument as a format string that indicates the color and line type of the plot. A list of markers is given at `http://matplotlib.org/api/markers_api.html#module-matplotlib.markers`. For example: `plot(x,y,'r+')` means that red plus signs (+) will be placed for each data point.

Marker Abbreviation	Marker Style
.	Point
,	Pixel
o	Circle
v	Triangle down
<	Triangle left
>	Triangle right
1	Tripod down
2	Tripod up
3	Tripod left
4	Tripod right
s	square
p	pentagon
*	star
h	hexagon
H	Rotated hexagon
+	plus
x	cross
D	Diamond
d	Thin diamond
-	Horizontal line

- **Line width**: Line width defines the width of markers.

  `linewidth=n` where `n` can be set as an integer, sets the marker size to a desired dimension.

- **Line style**: Line style defines the style of lines which connect the markers. They can be set off when data points need not be connected.

  `linestyle = '.'` sets the line style as a connecting dot between two data points. Similarly a number of other line style also exist.

Style Abbreviation	Style
-	solid line
–	dashed line
-.	dash dot line
:	dotted line

- **Color**: Color of markers can also be used for distinguishing data points belonging to two or more different data sets, but this method cannot be used where data needs to be published in Black and White color scheme.

```
plot(arange(10,100,1), linestyle='--', marker='+', color='g')
```

Above command sets the line style as `--`, markers as `+` in green color. A shortcut command would have been

```
plot(range(10), '--g+')
```

Following is the list of codes for choosing particular color:

Color Abbreviation	**Color Name**
b	blue
c	cyan
g	green
k	black
m	magenta
r	red
w	white
y	yellow

Apart from using above pre-defined symbols to choose a color, one can also use hexadecimal string such as `#FF00FF`, RGBA tuple like `(1,0,1,1)` and setting grayscale intensity as string like `'0.6'`.

- **Grid**: Grid can be turned off or an for a graph using the syntax:

  ```
  grid(True)
  ```

- **Legends**: Legends are used to differentiate between different types of data points from multiple graphs in a same figure by showing symbol for data type and printing text for the same.

  Their usage is illustrated at

  ```
  http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.
  legend
  ```

  By default, `legend()` takes input as the string provided within `plot()` function under the flag `label=''`. The location is set to be top-right corner by default. It can be changed as per requirement by setting `loc=` argument.

The URL at `http://matplotlib.org/api/lines_api.html#matplotlib.lines.Line2D` gives a variety of options for setting the line properties for a 2D `plot()` functions. Code `sqPlot3.py` shows a plotting of a formatted figure using a variety of formatter arguments.

```python
# import pylab and numpy
from matplotlib import pylab as pl
import numpy as np

# Create a figure of size 9x7 inches, 100 dots per inch
pl.figure(figsize=(9, 7), dpi=100)

# Create a new subplot from a grid of 1x1
pl.subplot(1, 1, 1)

# We wish to plot sin(x) and sin(2x)

# first we define x- axis in terms of pi units

X = np.linspace(-np.pi * 2, np.pi *2, 10e4, endpoint=True)

''' x-axis is defined from -2pi to 2pi with 10000 points
where last point is also included'''

S, S2 = np.sin(X), np.sin(2*X)

# Plot sin(x) with a blue continuous line of width 1 (pixels)
# labelled as sin(x)
pl.plot(X, S, color="blue", linewidth=1.0, linestyle="-", label
    = "$sin(x)$")

# Plot sine(2x) with a red continuous line of width 1 (pixels)
# labelled as sin(2x)
pl.plot(X, S2, color="red", linewidth=1.0, linestyle="-", label
    = "$sin(2x)$")

# Set x limits from -2pi to 2.5*pi
pl.xlim(-2* np.pi, 2.5* np.pi)

# Set x ticks
pl.xticks(np.linspace(-2.5 * np.pi, 2.5 * np.pi, 9, endpoint=
    True))

# Set y limits from 1.2 to -1.2
```

```
pl.ylim(-1.2, 1.2)

# Set y ticks
pl.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Set the tile as 'Sine waves'
pl.title('$sin(x)$ and $sin(2x)$ waves')

# Setting label on x-axis and y-axis

pl.ylabel('$sin(x)$ and $sin(2x)$')
pl.xlabel('$x$')

# Setting the grid to be ON
pl.grid(True)

# To show a legend at one corner for differentiating two curves
pl.legend()

# Show result on screen
pl.show()
```

sqPlot3.py

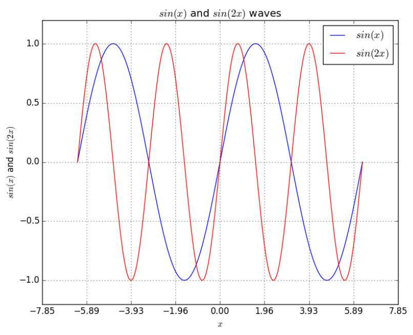


Figure 6.4: Plotting $sin(x)$ and $sin(2x)$

6.4 Setting up to properties

The `setup()` and `getup()` objects allows to set and get properties of objects. They work well with `matplotlib` objects. Hence for the object `plot()`, `setup()` can be used to set the properties.

6.5 Histograms

Histograms use vertical bars to plot events occurring with a particular range of frequency (called bins). The can be simply plotted using `hist()` function as in code `plottingHistogram.py`

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

a = np.random.rand(50)
pt.hist(a)
pt.show()
```

plotHistogram.py

The result in shown in figure 6.5. It is important to note that since we used random numbers as input to the `hist()` function, a different plot will be produced each time.

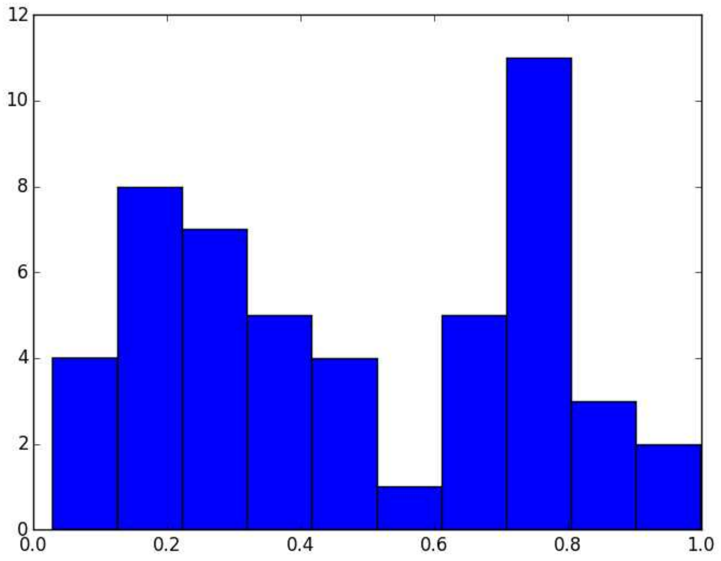


Figure 6.5: Histogram plot for 50 random numbers

Number of bins can be set to a number along with input variable as shown in code `plotHistogramBins.py`.

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

a = np.random.rand(50)
pt.hist(a,25) # setting number of bins to 25
pt.show()
```

plotHistogramBins.py

The result in shown in figure 6.6.

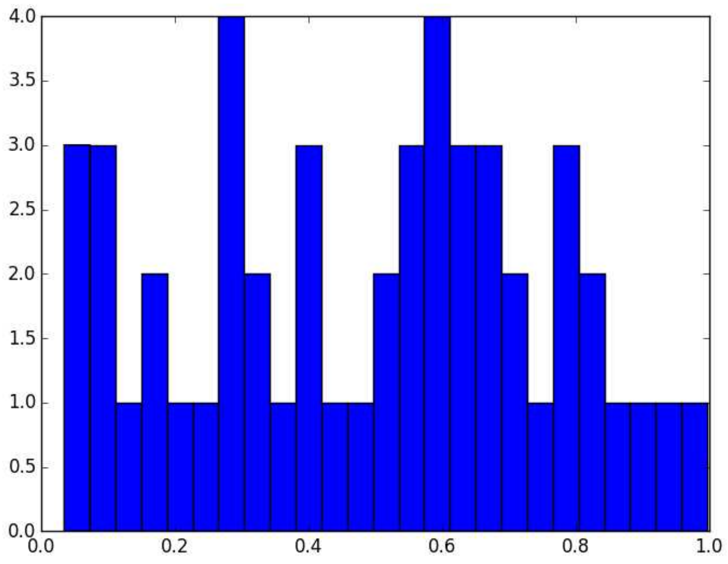


Figure 6.6: Histogram plot for 50 random numbers

6.6 Bar charts

One of simplest plots is to plot rectangular bars (either horizontally or vertically) where height of rectangle is proportional to the data value. This kind of graph is called a bar chart. These are generated by `bar()` function which takes two inputs for defining x-axis and y-axis, as opposed to `hist()` function which takes only one input. A sample code is presented in `bar.py` where x and y arrays are defined.

```
import matplotlib.pyplot as pl
import numpy as np
x = np.array([1,2,3,4,5,6,7,8,9,0])
y = np.array([1,4,2,3,4,5,7,6,8,7])
pl.bar(x, y)
pl.title('Vertical Bar chart')
pl.xlabel('$x$')
pl.ylabel('$y$')
pl.show()
```

bar.py

The result in shown in figure 6.7.

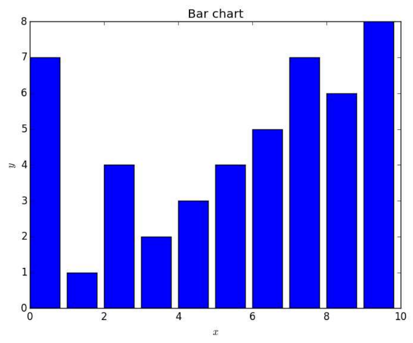


Figure 6.7: Vertical Bar chart

Bar charts and histograms look very similar. Difference lies in the way one defines them. Whereas `bar()` requires both x-axis and y-axis arguments, `hist()` requires only y-axis argument. `barh()` function plots horizontal bars.

```
import matplotlib.pyplot as pl
import numpy as np
x = np.array([1,2,3,4,5,6,7,8,9,0])
y = np.array([1,4,2,3,4,5,7,6,8,7])
pl.barh(x, y)
pl.title('Horizontal Bar chart')
pl.xlabel('$x$')
pl.ylabel('$y$')
pl.show()
```

barh.py

The result in shown in figure 6.8.

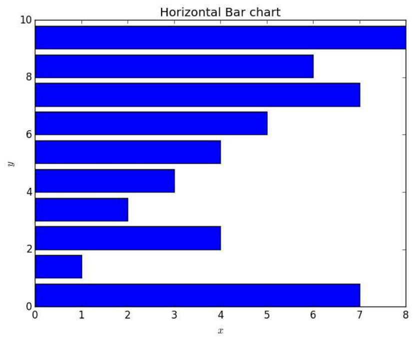


Figure 6.8: Horizontal Bar chart

6.7 Error Bar Charts

All of experimental scientific work involves errors and it is important to plot errors along with the data for many reasons. Errors must be plotted so that one can judge data quality, one can understand the regions of data where error is huge or minuscule etc. In `matplotlib`, the `errorbar()` function enables one to create such graphs called error bar charts. The representation of distribution of data values is done by plotting a single data point, (commonly) the mean value of dataset, and an error bar to represent the overall distribution of data. To accomplish this, code `ploterror.py` defined an array for x axis and then defined an array for y axis using the formula $y = x^2$. Error is stored in third array saved with variable name `err`. These variable names are passed as arguments to `errorbar()` function hence it takes three variable in a in a sequence as `x,y,err`.

```
import matplotlib.pyplot as pl
import numpy as np
x = np.arange(0, 4, 0.2) # generated data point from 0 to 4 with
    step of 0.2
y = x*2 # y = e^(-x)
err = np.array([
    0,.1,.1,.2,.1,.5,.9,.2,.9,.2,.2,.2,.3,.2,.3,.1,.2,.2,.3,.4])
pl.errorbar(x, y, yerr=err, ecolor='r')
pl.title('Error bar chart with symmetrical error')
pl.xlabel('$x$')
pl.ylabel('$y$')
pl.show()
```

ploterror.py

The file produces the graph shown in figure 6.9.

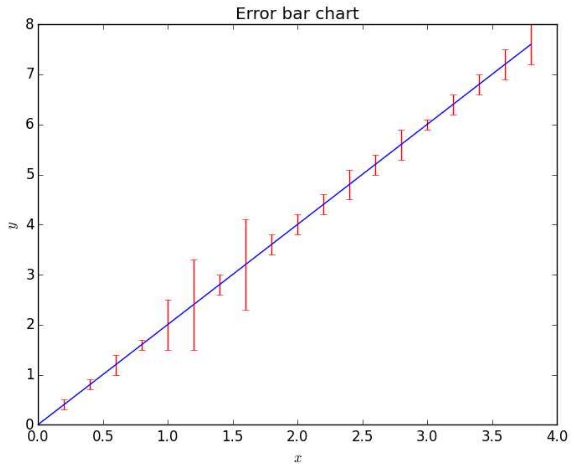


Figure 6.9: Symmetrical Error bar chart

`ecolor` sets the color for error bars. Just as setting value for `yerr` keyword, one can also set `xerr` to produce error bars on x-axis too.

In example `ploterror.py` error bars are symmetrical i.e positive and negative errors are equal. For plotting asymmetrical error bars, `errorbar()` would incorporate two arrays for error definition as follows.

```
import matplotlib.pyplot as pl
import numpy as np
x = np.arange(0, 4, 0.2) # generated data point from 0 to 4 with
    step of 0.2
y = x*2 # y = e^(-x)
err_positive = np.array([
    0.5,.1,.1,.2,.1,.5,.9,.2,.9,.2,.2,.2,.3,.2,.3,.1,.2,.2,.3,.4])

err_negative = np.array([
    0.2,.4,.3,.1,.4,.3,.1,.9,.1,.3,.5,.0,.5,.1,.2,.6,.3,.4,.1,.1])

pl.errorbar(x, y, yerr=[err_positive, err_negative], ecolor='r')
pl.title('Error bar chart with Asyymetric error')
pl.xlabel('$x$')
pl.ylabel('$y$')
pl.show()
```

ploterror1.py

The file produces the graph shown in figure 6.10.

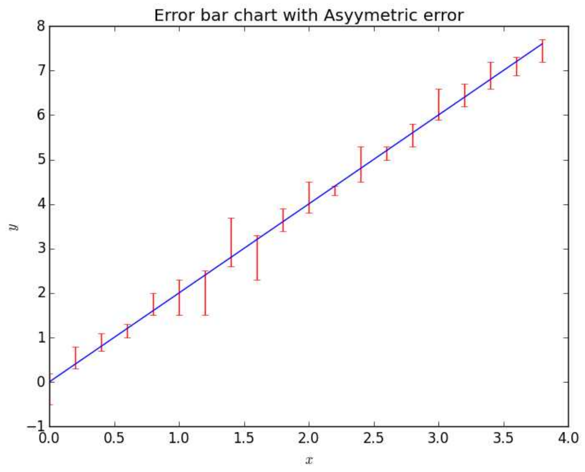


Figure 6.10: Asymmetrical Error bar chart

6.8 Scatter plot

Scatter plot is simply points plotted on a 2D mesh (made of two axes, say x and y). The data aren't connected with lines, hence they look scattered unconnected!

This is achieved by `scatter()` function which takes two arrays as arguments. Scatter plots are used to get correlation between two variables. When plotted, the clusters show strong correlation between particular data ranges. This is one of the key actions required by regression analysis.

```
import matplotlib.pyplot as pl
import numpy as np
x = np.random.rand(1000)
```

```
y = np.random.rand(1000)
pl.scatter(x,y)
pl.title('Scatter Chart')
pl.xlabel('$x$')
pl.ylabel('$y$')
pl.show()
```

scatter.py

The file produces the graph shown in figure 6.11.

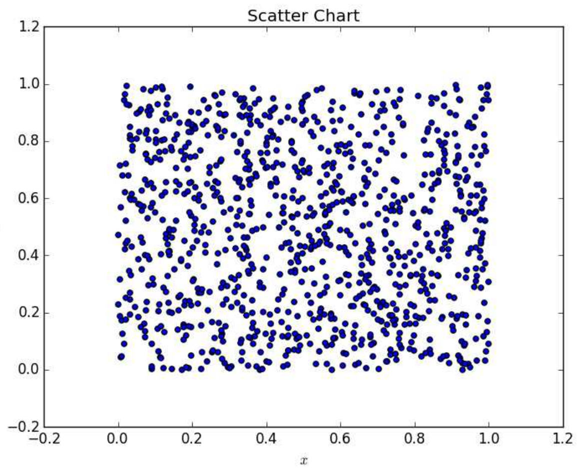


Figure 6.11: Scatter Plot

6.9 Pie Chart

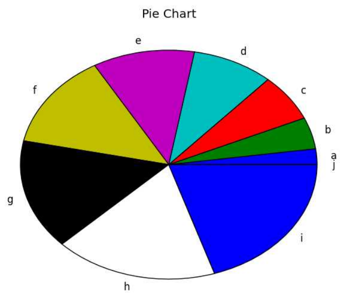
When data needs to be categorized into sectors for number of events in a particular range, pie charts come handy. Pie charts are circular shapes where sectors/wedges are carved out for different data ranges where size of a wedge is proportional to the data value. The `pie()` function works in this regard.

```
import matplotlib.pyplot as pl
import numpy as np
x = np.array([1,2,3,4,5,6,7,8,9,0])
label = ['a','b','c','d','e','f','g','h','i','j']
explode = [0.2, 0.1, 0.5, 0, 0, 0.3, 0.3, 0.2, 0.1,0]
pl.pie(x, labels=label, explode = explode, shadow=True, autopct=
    '%2.2f%%')
```

```
pl.title('Pie Chart')
pl.show()
```

pie.py

The file produces the graph shown in figure 6.12.



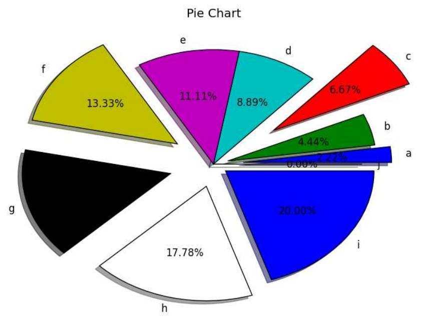


Figure 6.12: Unexploded and exploded Pie Chart

When `explode` label is not defined, one gets an unexploded pie chart. `autopct` sets the show the percentage of weight for a particular weight which can be set by format specifier. `%2.2f%%` sets the display of percentage weights uptill 2 decimal places with 2 significant digits. `shadow` provides a shadow below the wedge so that it looks like a real pie!

6.10 Polar Plots

Until now, all plots have been dealing with data defined in cartesian system. For data defined in polar system i.e (r, θ) instead of (x, y). Polar plots are obtained by `plot()` function.

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

r = np.arange(0, 10.0, 0.1)
theta = 2* np.pi * r

pl.polar(theta, r, color ='g')
pl.show()
```

polar.py

The file produces the graph shown in figure 6.13.

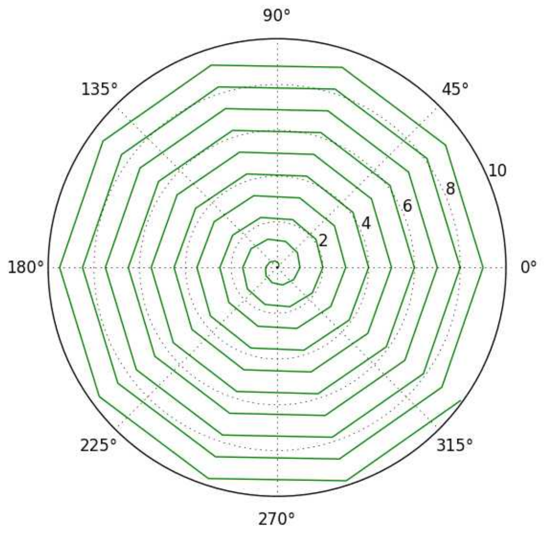


Figure 6.13: Polar Plot

6.11 Decorating plots with text, arrows and annotations

Sometimes, one is required to put text at a specific place on the plot (say coordinated (x, y)). `text()` function is used to place text as shown below.

```
import matplotlib.pyplot as pl
```

```python
import numpy as np
x = np.arange(0, 2*np.pi, .01)
y = np.sin(x)
pl.plot(x, y, color = 'r');
pl.text(0.1, -0.04, '$sin(0) = 0$')
pl.text(1.5, 0.9, '$sin(90) = 1$')
pl.text(2.0, 0, '$sin(180) = 0$')
pl.text(4.5, -0.9, '$sin(270) = -1$')
pl.text(6.0, 0.0, '$sin(360) = 1$')
pl.annotate('$sin(theta)=0$', xy=(3, 0.1), xytext=(5, 0.7),
    arrowprops=dict(facecolor='green', shrink=0.05))
pl.title('Inserting text and annotation in plots')
pl.xlabel('$theta$')
pl.ylabel('$y = sin( theta)$')
pl.show()
```

textPlot.py

The file produces the graph shown in figure 6.14.

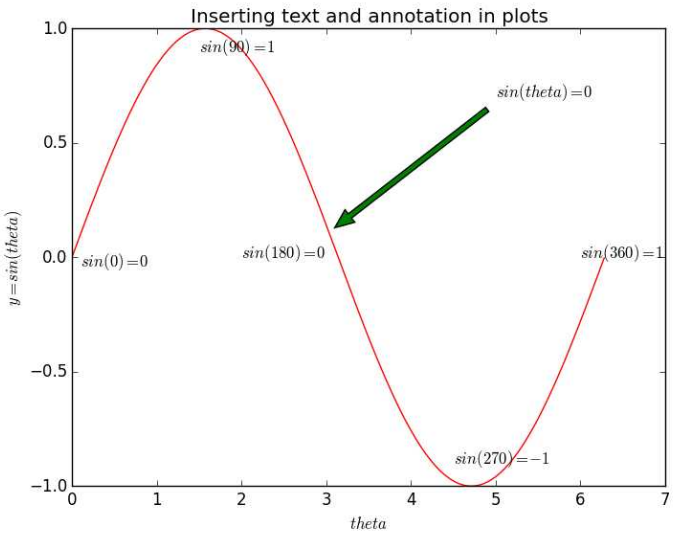


Figure 6.14: Inserting text in the Plot

As seen in the example `textPlot.py`, text, arrow and annotations can be placed at appropriate places by defining coordinate axis pints for them. A convenient method of identifying appropriate coordinates is to roll over the mouse on body of plot and look for down-left corner of the figure where present mouse coordinates are shown. `help(matplotlib.pyplot.annotate)` gives useful inputs to use this function.

6.12 Subplots

Multiple plots can be plotted using subplot option where different plots are considered to be a matrix of graphs. Just like a regular matrix, elements are identified by index. As seen in code `subplot1.py` subplots are located using indices like (222) which essentially mean that the matrix is a 2×2 and one is accessing 2^{nd} position to place the `scatter()` function based plot. Similarly (221) uses a `plot()` function at 1^{st} position, (223) at 3^{rd} position plots a histogram using `hist()` function and (224) is 4^{th} plot using `barh()` giving a horizontal bar graph.

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

x = np.arange(10)
y1 = np.random.rand(10)
y2 = np.random.rand(10)

fig = pl.figure()


ax1 = fig.add_subplot(221)
ax1.plot(x,y1)

ax2 = fig.add_subplot(222)
ax2.scatter(x,y2)

ax3 = fig.add_subplot(223)
ax3.hist(y1)

ax4 = fig.add_subplot(224)
ax4.barh(x,y2)

pl.show()
```

subplot1.py

The file produces the graph shown in figure 6.15.

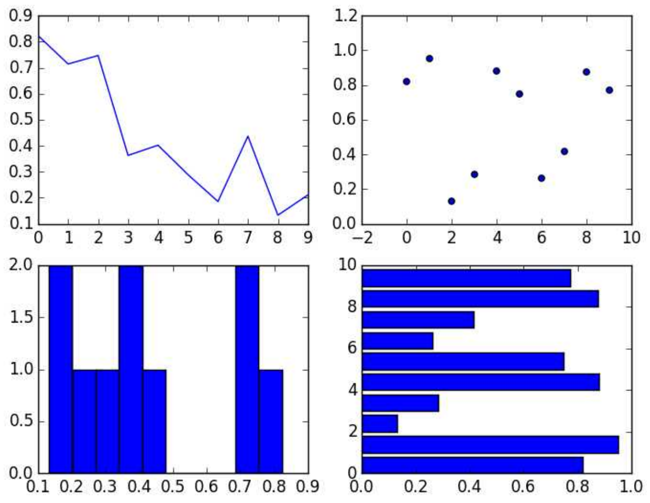


Figure 6.15: Subplots

6.13 Saving plot to a file

Most of the times, a graph plotted using `matplotlib` needs to be saved for future reference and use. In these cases, `savefig()` function is used to specify the file name, permission, file type etc. for the figure. As an example, code `sqPlot4.py` results a file named `plot1.png` in the working folder. (Those who are working with Ipython can know about present working directory by typing `pwd`). If one provides a proper path as the string argument of `savefig()` function, a file is created at that path provided user have proper privileges to create the same.

```python
import matplotlib.pyplot as pt
import numpy as np
pt.plot(np.arange(10))
pt.savefig('plot1.png')
```

sqPlot4.py

The file produces the graph shown in figure 6.16.

It is important to know the size of file and resolution of figure for the purpose of publication at both off-line/print and on-line medium. When

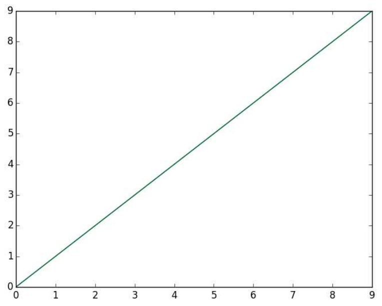


Figure 6.16: Plotting $sin(x)$ and $sin(2x)$

none of the arguments for setting the resolution of figure and size of file is set within the program, default values are used. These default values can be known by following code:

```
>>>import matplotlib as ml
>>>ml.rcParams['figure.figsize']
[8.0, 6.0]
# Default figure size is 8 X 6 inches
>>>ml.rcParams['savefig.dpi']
100.0
# Default figure resolution is 100 dpi
```

Since a 8×6 inches figure is created with 100 dpi (dots/pixels per inch), hence a 800×600 pixels image is saved using `savefig()` by default. When this file is directed towards a computer graphic terminal for displaying, length units are ignored and pixels are displayed. If file is directed towards a printing media like a printer or plotter, lengths parameters and DPI determine figure size and resolution.

6.14 Displaying plots on web application servers

In the era of connecting using internet, many problems require plotting interactive graphs on web pages. These require different types of plots to be plotted on web pages written in different kinds of languages working under different environments. It is important to note that `matplotlib` requires graphic user interface requiring a X11 connection. Hence it is important to turn on this faculty on a web application server before updating the plots dynamically, generated by `matplotlib`. There are two aspects to plotting graphs on computer in general. Coding using a set of commands to make a script is known as **frontend** task which require a **backend** effort. Backend does all the hardwork of interacting with graphical capabilities of the system to produce a graph in a desired plot. Plots can be plotted interactively using backends like `pygtk`, `wxpython`, `tkinter`, `qt4`, `macosx` or they can be plotted non-interactively (permanently saved as files on computer) using backends like `PNG`, `SVG`, `PS`, `PDF`. The latter are also known as *hardcopy* backends.

There are two routes to configure a backend.

1. `matplotlibrc` file in installation directory can be edited to set `backend` parameter to a particular value such as one example below:

```
backend : WXAgg   # use wxpython with antigrain (agg)
  rendering

```

   `matplotlibrc` is present at `/etc/matplotlibrc` on a LINUX system

2. Using `use()` to set a particular backend temporarily.

```
  >>>import matplotlib
  >>>matplotlib.use('PDF')   # generate PDF file as
  output by default

```

Choosing one of the ways to set backend depends on the application of the program. If the program aims to save plots as non-interactive `PDF` files temporarily, method number 2 can be used, which works until another program sets the backend differently. Setting the backend must be done before `import matplotlib.pyplot` or `import matplotlib.pylab`. More on using various backends for variety of application can be found on `http://matplotlib.org/faq/usage_faq.html#what-is-a-backend`. For a web application server, setting backend as `WXAgg, GTKAgg, QT4Agg, TkAgg` will work.

One way to save transparent figures as opposed to white colored by default is to set `transparent=TRUE`. This is particularly important in the case when figures needs to be embedded on a web-page with predefined-background color/image.

Another way of turning interactive mode to be ON or OFF is by including commands `matplotlib.pyplot.ion()` and `matplotlib.pyplot.ioff()` respectively.

Ipython notebook

While working with `Ipython`, if one wishes to work with plots to dynamically change by issuing commands at `Ipython` command prompt, then one simply issues a command at UNIX terminal:

```
$ ipython −pylab
```

This enables a special `matplotlib` support mode in `IPython` that looks for configuration file looking for the backend, activating the proper GUI threading model if required. It also sets the `Matplotlib` interactive mode, so that `show()` commands does not block the interactive `Ipython` shell.

In case of working with `Ipython notebook` environment, issuing the command:

```
$ ipython notebook() −−pylab=inline
```

produces the graphs in-line i.e. within the body of the code in between the command lines where plot is called, otherwise plots pop out in a separate window. In-line mode is useful while designing teaching material. But before sharing with concerned person, it should be ensured that encoded backends and dependencies are installed on users computer.

6.15 Working with matplotlib in object mode

Pythonic way of using matplotlib is to use it in object mode where explicit definition of an object allows ultimate customization. For this purpose, one must define each element of a graph as an object and use the properties to customize the same. The hierarchy of three basic objects used for the purpose is as follows:

1. `FigureCanvas`: Container class for `Figure` instance
2. `Figure`: Container class for `axes` instance
3. `Axes`: Axes are the rectangular areas to hold various plot features like lines, ticks, curves, text etc.

When working with matplotlib in object-oriented mode, one specifies a `FigureCanvas` which holds `figure(s)` which in turn holds `axes` where variety of plotting features can be implemented. The whole process allows customization to any extent as can be imagined. The following example explain this concept.

```python
import matplotlib.pyplot as plt
fig = plt.figure()
# variable fig stores "figure" instance
ax1 = fig.add_subplot(221)
# variable ax1 stores the subplot of figure at 1st place in 2 x
    1 matrix
ax1.plot([-1, 1, 4], [-2, -3, 4]);
# ax1 iscalled and plot fucntion is given to it.
# plot function carries two lists giving x and y axis points for
     graph
ax2 = fig.add_subplot(222)
ax2.plot([1, -2, 2], [0, 0, 2]);
# same logic as for ax1
ax3 = fig.add_subplot(223)
ax3.plot([10, 20, 30], [10, 20, 30]);
ax4 = fig.add_subplot(224)
ax4.plot([-1, -2, -3], [-10, -20, -30]);
plt.show()
# show the figure on computer terminal
```

objPlot.py

The resulting plot is given by 6.17

Most of the plots in present book will be plotted in objective mode henceforth.

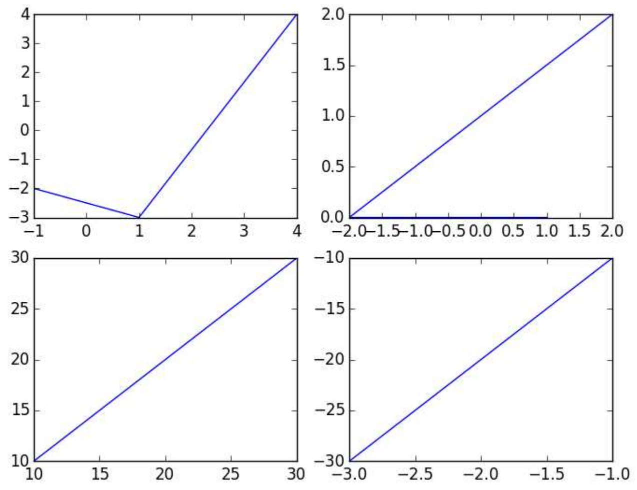


Figure 6.17: Plotting using object mode capabilities

6.16 Logarithmic plots

A variety of engineering data uses logarithmic scales, particularly those where changes result in an order of magnitude change in values of observed variable. Python provides facility to plot logarithmic plots.

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

x = np.arange(0., 10, 0.01)

fig = plt.figure()

ax1 = fig.add_subplot(221)
y1 = np.log(x)
ax1.plot(x, y1);
ax1.grid(True)
ax1.set_ylabel('$y = log(x)$');
ax1.set_title('y-axis in log scale')

ax2 = fig.add_subplot(222)
y2 = np.sin(np.pi*x/2.)
ax2.semilogx(x, y2, basex = 3);
ax2.grid(True)
ax2.set_title('x-axis in log scale')
```

```

ax3 = fig.add_subplot(223)
y3 = np.sin(np.pi*x/3.)
ax3.loglog(x, y3, basex=2);
ax3.grid(True)
ax3.set_ylabel('both axes in log');

ax4 = fig.add_subplot(224)
y4 = np.cos(2*x)
ax4.loglog(x, y3, basex=10);
ax4.grid(True)

plt.show()
```

log.py

Following axes instances are defined:

1. `ax1` uses $y = log(x)$
2. `ax2` uses $y = sin(\frac{\pi x}{2})$
3. `ax3` uses $y = sin(\frac{\pi x}{3})$
4. `ax4` uses $y = cos(2x)$

The resulting graph is given by Figure: 6.18

As seen in the program `log.py` logarithmic values can be directly passed to `plot()` function. When a particular axis needs to be plotted in logarithmic values then `semilog()` and `semilogy()` can be used where a base index can also be defined. The value of that particular axis is converted into logarithmic scale and plotted subsequently. When logarithmic values needs to be plotted on both axes `loglog()` functions needs to be invoked.

Logarithmic plots find their use in a variety of fields like signal processing, thermodynamics etc. Essentially, whenever data changes by an order of a magnitude, its easier to observe it using logarithmic plot. Logarithmic scale is a non-linear scale. The ability of change base of logarithmic function, provides a powerful tool at the hands of developers to plot creatively to derive most meaningful conclusion.

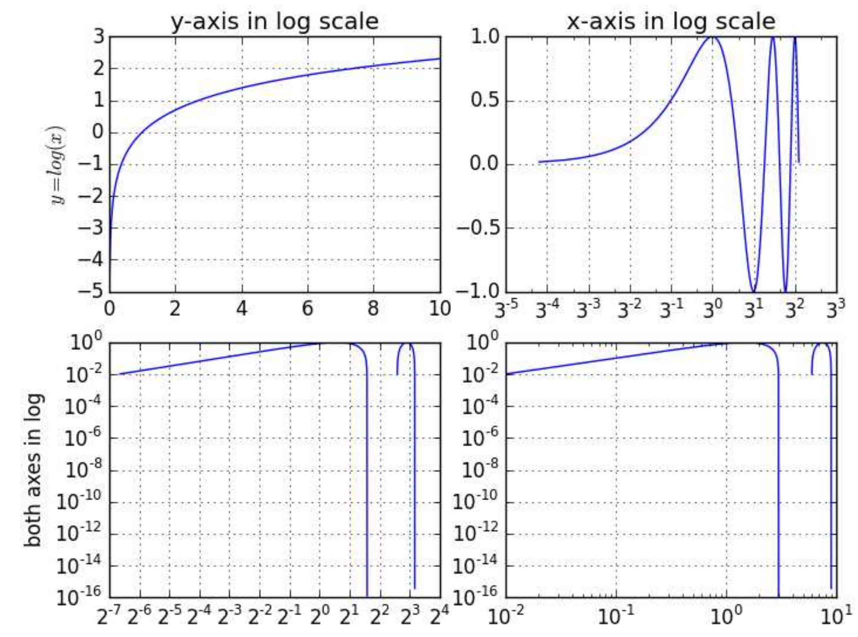


Figure 6.18: Plotting using object mode capabilities

6.17 Two plots on same figure with atleast one axis different

Using functions `twinx()` and `twiny()` one can use two x or y axes on the same figure to plot two sets of data points. An example to use `twinx()` is used here where x axis is *twinned* to produce plots of two data sets sharing same x-axis data points.

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

x = np.arange(0., 100, 1);
y1 = x**2;
# y1 is defined as square of x values
y2 = np.sqrt(x);
# y2 is defined as square root of x values

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.plot(x, y1, 'bo');
```

```
ax1.set_ylabel('$x^{2}$');
ax2 = ax1.twinx() # twinx() function is used to show twinned x
    axes
ax2.plot(x, y2, 'k+');
ax2.set_ylabel('$\sqrt{x}$');
ax2.set_title('Same x axis for both y values');
plt.show()
```

twinx.py

It is worth noting that two different axes instances namely `ax1` and `ax2` are **superimposed** on each other where data from `y1` being alloted to axes instance `ax1` and data from `y2` being alloted to axes instance `ax2`. This also illustrates the power of defining a plot in object mode. The corresponding plot is given in 6.19

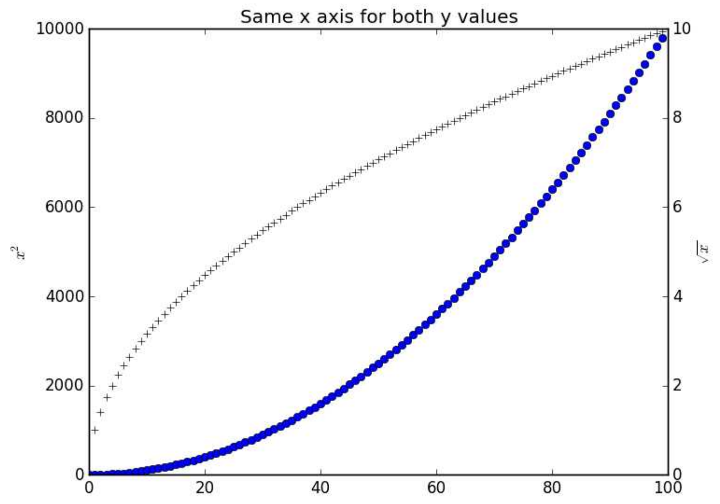


Figure 6.19: Using twinx() variable

6.18 Contour plots

In some engineering applications, contour plots become an essential part of interpretation because they can define segregation of data into regions based on certain similarity. For example, if one imagines a mountain viewed from top, one can define regions of similar height being shown with a closed loop. Thus a mountain will be a series of loops. Similarly, a 2D map of non-uniformly heated region can be viewed as contours depicting regions of same temperature. A region of rainfall can be viewed as contour showing regions of dis-similar size of droplets.

Hence contour lines are also called *level lines* or *isolines*. The term *iso-* is attached to data points having constant value and the regions of these data points are separated by contours.

For a contour plot, one needs x, y and z axis where z axis defines the height. The data with same height is clubbed together within isolines.

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

# defining data for x, y, z axes
x = np.linspace(0,1,100)
y = np.linspace(1,2,100)
(X,Y) = np.meshgrid(x,y)
z = np.sin(X)-np.sin(Y)

# plotting contour
fig = plt.figure()

ax1 = fig.add_subplot(211)
c1 = ax1.contour(x,y,z)
l1 = ax1.clabel(c1)
lx1 = ax1.set_xlabel("x")
ly1 = ax1.set_ylabel("y")


# plotting filled contour
ax2 = fig.add_subplot(212)
c2 = ax2.contourf(x,y,z)
l2 = ax2.clabel(c2)
lx2 = ax2.set_xlabel("x")
ly2 = ax2.set_ylabel("y")

plt.show()

# plotting filled contour
```

contour.py

The corresponding plot is given at 6.20

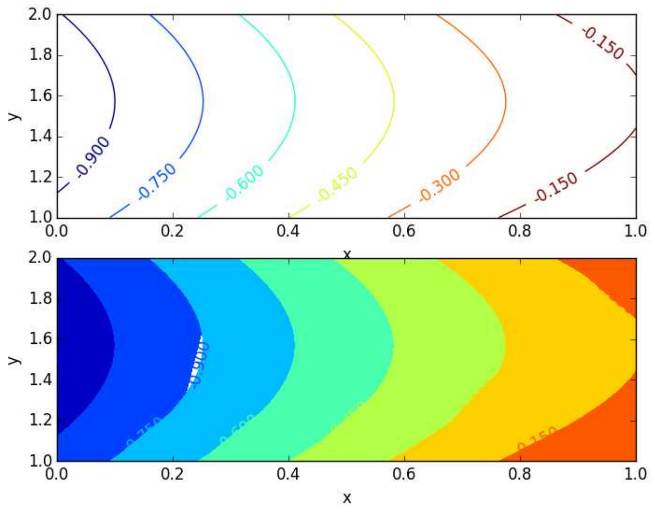


Figure 6.20: Contour plots

`contour()` and `contourf()` functions can be used to plot unfilled and filled contour. A simple command `help(contour)` gives extensive information about setting various parameters for a contour plot.

6.19 3D plotting in matplotlib

With advanced in computing technologies at both software and hardware's end, it has become easier to produce interactive 3D plots on graphic terminals. `matplotlib` provides a decent number of options in this regard, which are discussed in following subsections.

6.19.1 Line and scatter plots

`matplotlib`'s `toolkits` has a class `mplot3D` which provides `Axes3D` object. Using the `projection='3D'` keyword, an `Axes3D` object is created which provides the screen area to show a 3D plot. `Axes3D` can then be passed on to a `figure` object to show it as a figure. A line plot can be simply created by passing three arguments to `plot()` function as seen in `3Dline.py` python script.

```python
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
```

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

fig = plt.figure()
ax = fig.gca(projection='3d')

x = np.linspace(-10*(np.pi),10*(np.pi),10e4)
y = np.sin(x)
z = np.cos(x)

ax.plot(x, y, z, label='$y=sin(x)$ and $z = cos(x)$')
ax.legend()
ax.set_title('3D line curve')
ax.set_xlabel('$x$')
ax.set_ylabel('$y = sin(x)$')
ax.set_zlabel('$z = cos(x)$')
plt.show()
```

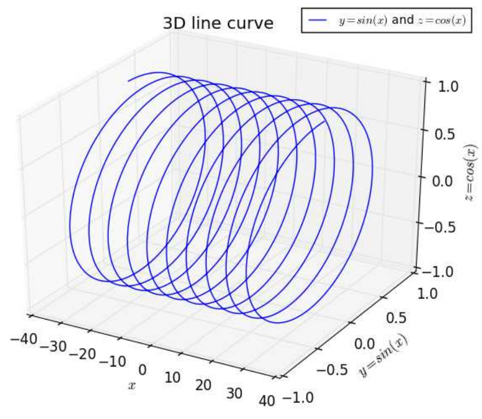
3Dline.py

```python
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x = np.linspace(-5*(np.pi), 5*(np.pi),200)
y =np.sin(x)
z =np.cos(x)

ax.scatter(x, y, z, marker='*')

ax.set_xlabel('$x$')
ax.set_ylabel('$y = sin(x)$')
ax.set_zlabel('$z = cos(x)$')
ax.set_title('Scatter plot in 3D')

plt.show()
```

3Dscatter.py

The corresponding graph is shown in figure 6.21

Scatter plots are plotted in 3D in similar way as line plots. This is illustrated in the python scripy `3Dscatter.py` and the corresponding graph is shown in figure 6.21



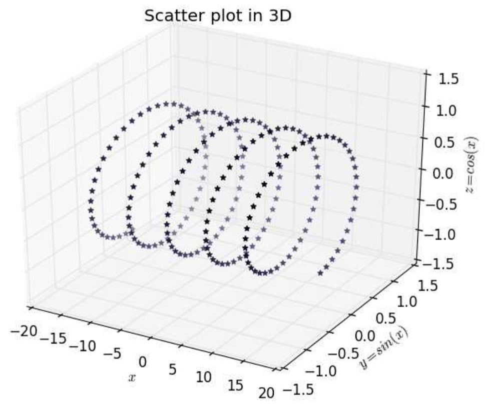


Figure 6.21: line and scatter plot in 3D

6.19.2 Wiremesh and Surface plots

During computation with discrete values, it is sometimes useful to plot a wiremesh plot as seen in figure 6.22. `meshgrid` function is used to generate a grid of points using x and y values. Function

$$z = \sqrt{x^2 + y^2}$$

is generated on top of this grid and plotted using wiremesh function. `rstride` and `cstride` define the row and column step size.

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

a = np.arange(-5, 5, 0.25)
b = np.arange(-5, 5, 0.25)
x, y = np.meshgrid(a, b)
z = np.sqrt(x**2 + y**2)

ax.plot_wireframe(x, y, z, rstride=2, cstride=2)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_zlabel('$z = \sqrt{x^{2}+y^{2}}$')
ax.set_title('Wiremesh type of 3D plot')

plt.show()
```

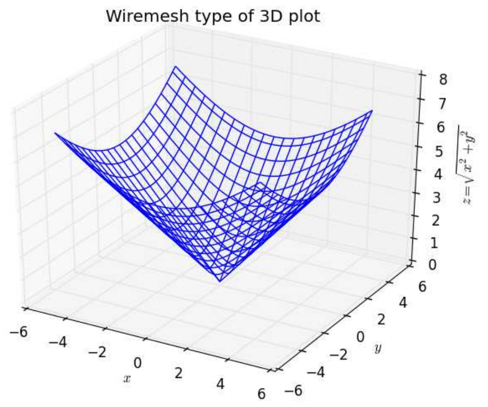
3Dwiremesh.py

Surface plots are similar to wiremesh plots except the fact that its continuously filled up. Hence instead of `wiremesh()` function, one uses `surface()` function.

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

a = np.arange(-5, 5, 0.25)
b = np.arange(-5, 5, 0.25)
x, y = np.meshgrid(a, b)
z = np.sqrt(x**2 + y**2)

ax.plot_surface(x, y, z, rstride=2, cstride=2)

ax.set_xlabel('x')
ax.set_ylabel('$y$')
ax.set_zlabel('$z = \sqrt{x^{2}+y^{2}}$')
ax.set_title('Surface type of 3D plot')

plt.show()
```

3Dsurface.py

The corresponding graph is shown in figure 6.22



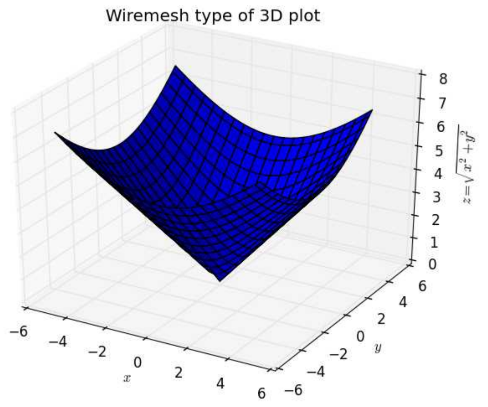


Figure 6.22: Wiremesh and surface plot

6.19.3 Contour plots in 3D

Just as in two dimensional contours, 3D contour plots employ *isosurfaces* i.e. surfaces having equal height. Using `contour()` and `contourf()` functions, one can plot unfilled and filled contour plots.

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
x = np.linspace(2*np.pi,-2*(np.pi),1000)
y = np.linspace(2*np.pi,-2*(np.pi),1000)
X,Y = np.meshgrid(x,y)
Z = np.sin(X) + np.sin(Y)

cont = ax1.contour(X, Y, Z)
ax1.clabel(cont, fontsize=9, inline=1)
ax1.set_xlabel('$x$')
ax1.set_ylabel('$y$')
ax1.set_title('Contour for $z=sin(x)+sin(y)$')

ax2 = fig.add_subplot(122, projection='3d')
Z = np.sin(X) + np.sin(Y)
cont = ax2.contourf(X, Y, Z)
ax2.clabel(cont, fontsize=9, inline=1)
ax2.set_xlabel('$x$')
ax2.set_ylabel('$y$')
ax2.set_title('Filled Contour for $z=sin(x)+sin(y)$')



plt.show()
```

3Dcontour.py

The corresponding graph is shown in figure 6.23

6.19.4 Quiver plots

```python
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(np.pi,-(np.pi),10)
y = np.linspace(np.pi,-(np.pi),10)
(X,Y) = np.meshgrid(x,y)
u = -15*X
v = 5*Y
q = plt.quiver(X,Y,u,v,angles='xy',scale=1000,color='b')
```

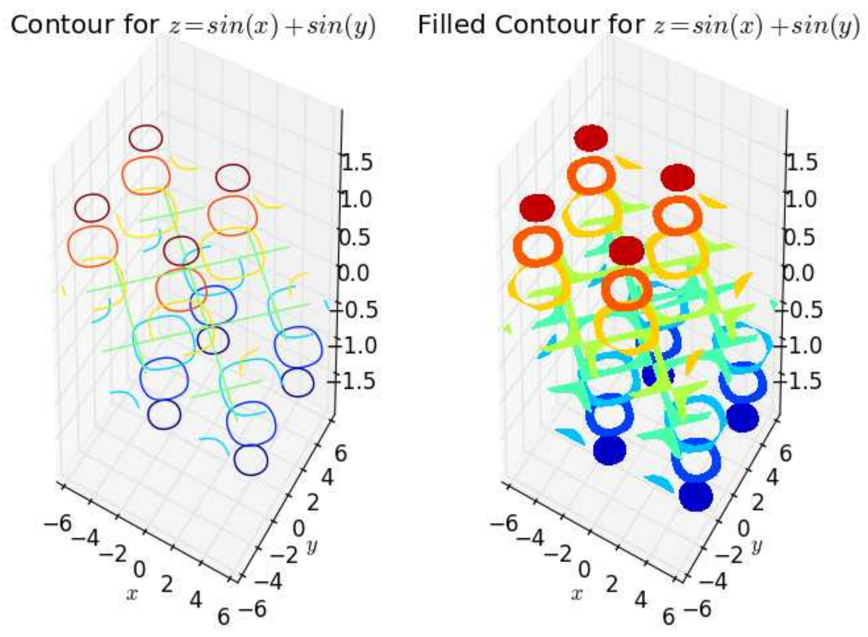


Figure 6.23: Contour plots using contour and contourf functions plot

```
#p = plt.quiverkey(q,1,16.5,50,"50 m/s",coordinates='data',color
    ='r')
xl = plt.xlabel("x (km)")
yl = plt.ylabel("y (km)")
plt.show()
```

3Dquiver.py

The corresponding graph is shown in figure 6.24

6.20 Other libraries for plotting data

`matplotlib` has been in use to such an extent that new developers do not realize other options to plot the data. There are variety of other ways to plot data in other modules which might have more powerful plotting capabilities depending on context. Amongst them are plotly, mayavi and

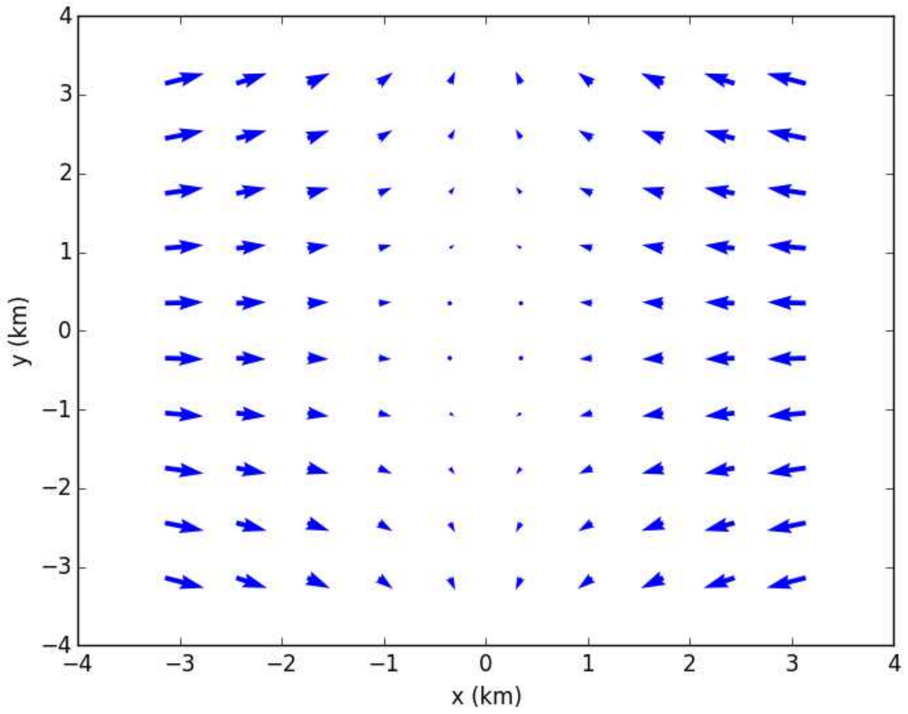


Figure 6.24: Quiver plots

gnuplot to name a few. A brief discussing follows about using plotly, which interestingly, plots the data on the web. This is particularly interesting for those engineering applications where the sensor data needs to be plotted on web in real time.

6.20.1 Plotly

Plotly is an on-line analytics and data visualization tool. Apart from python, plotly can also plot data used in Perl, Julia, Arduino, R and MATLAB®. Weblink `https://plot.ly/` gives pretty good idea about capabilities of plotly. Essentially plotly allows plotting and publishing the graph on-line for allowing collaboration. Hence one must be connected to internet before working with plotly library.

First, one needs to make a user account at plotly website. Account will provide a **username** and **API key** which needs to be used in the program. Then one can write a code either as one command at a time on python terminal or as a python script.

```
import numpy as np
import plotly.plotly as py
from plotly.graph_objs import *
py.sign_in('username','APIkey')
data = Data([Scatter(x=np.arange(100),y=np.random.randn(100),
    name='trace 0')])
fig = Figure(data=data)
plot_url = py.plot(fig)
```

plotly.py

User needs to provide the *username* and *API key* in the above script. The above script results in generating a scatter plot on-line at workspace in the user account. These graphs can be plotted quite interactively using the functions provided by plotly. When data is steamed from a device connected to a live sensor, live data is plotted in real time. This can further be embedded on a website.

6.21 Summary

The present chapter discussed various plotting options available while working with python. The ease of plotting data is one of the most attractive feature from python. Just a few lines of code visualize the data in a variety of ways. Visualization is the back bone of data presentation and analysis since understanding of data becomes more clearer. Apart from simple visualization, `matplotlib` allows rich feature to decorate the graph with useful information in a desired manner.

7 File I/O

7.1 Introduction

Handling files is an essential part of the process of computation. Python provides many features to perform this act. Files come in a variety of format and hence any programming language enjoying the capabilities of handling files must provides the functionalities for handling variety of file formats and opening, making, editing and deleting them as desired, with ease.

7.2 Reading input from keyboard

UNIX and alike systems treat all computing resources as files, which include computer's peripherals. Keyboard is one of them and reading keystrokes to input values into a program remains critical functionality to any programming language. Python provides two functions for the same.

7.2.1 input and raw_input

Two built-in functions namely `input` and `raw_input` provide the functionality to incorporate values provided through a keyboard. `raw_input` reads one-line and returns it as a string. on the other hand, `input` function, treats the input as a valid python expression nd returns the result.

```
String_from_raw_input = raw_input("Enter the input for raw_input
    function:")
print "raw_input yields = ", String_from_raw_input

String_from_input = input("Enter the input for input function:")
print "raw_input yields = ", String_from_input
```

input.py

The result is shown below:

```
Enter the input for raw_input function:Hi
raw_input yields =  Hi

Enter the input for input function:[x for x in range(10)]
raw_input yields =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

7.3 file object

Python has an in-built object namely **file** to treat data files. `help(file)` gives detailed information about the same. A data file needs to be *opened* before it can be dealt with in any way. Once opened, a file can be *read* for editing. Editing can be done by *writing* to the file some data. After these actions, the file needs to be *closed*. Two methods of `file` object is shown in table 7.1.

`open()`	Open a file
`close()`	Close a file

Table 7.1: Methods for file object

`open()` function opens a file using the syntax.

```
file_object  = open ( file_name [,access_mode][,buffering])
```

For example, a file `a.txt` can be opened as:

```
file1 = open("a.txt", "r")
```

Here `file1` variable will store the *file object* containing the contents of the file named `a.txt`. This file is opened in *read only* mode as specified by second parameter given by the string `"r"`. A number of modes exist for files to be opened, as shown in table 7.2

`r`	read only
`rb`	read only in *binary format*
`r+`	read and write
`rb+`	read and write in *binary format*
`w`	write only
`w+`	read and write
`wb+`	read and write in *binary format*
`a`	append
`a+`	append and read
`ab+`	append and reading in *binary format*

Table 7.2: Different modes in which a file can be opened

1. In the mode `w,wb,wb+`, python program over-writes a file if it exists. If it does not exist then creates a new file for writing.
2. In the mode `a,ab,ab+`, python program appends the contents of a file if it exists. If it does not exist then creates a new file for writing.

7.4 file object's attributes

Different methods to access various attributes of a file object are provided to a programmer for convenience of knowing the state of file operation. Some of them are discussed below.

`file.name`	Returns the name of the file
`file.closed`	Returns boolean object `True/False` if file is closed/open
`file.mode`	Returns access mode at the time of opening the file

Table 7.3: attributes of file object

To understand these concept in better manner, following exercise can be done. Under a new directory of choice, a file named `a` is created.

```
>>>a1 = open("a",'w')
```

This command will make a new file named `a` in working directory and open it in *write* mode. This file object is stored in a variable named `a1`.

```
1  >>>type(a1)
2  file
3
4  >>>a1.name
5  'a'
6
7  >>>a1.closed
8  False
9
10 >>>a1.mode
11 'w'
12
13 >>>a1.close()
14 >>>a1.closed
15 True
16
17 >>>a1.mode
18 'w'
19
20 >>>a1.name
21 'a'
```

Line number 1 inquired about the type of object stored in the variable name `a1` which is returned as a `file` type. Line 4 inquires about name of the file which is returned as `a`. Line 7 inquires if the file is closed. Since the present set of commands have not closed the file so a boolean data type `False` is returned. Line number 10 inquires about the mode of file, for which *write* is returned. When the file is closed using instructions at Line 13 then the inquiry about its closing status at Line number 14 is returned as a boolean object `True`. All attributes can still be inquired after closing as seen in Line 17 and 20.

7.5 Reading and writing to files

The `write()` and `read()` methods enable writing and reading the **open** files. If the files are closed then they will show error. `write()` takes a string as input argument.

```
>>>a2 = open('fun.txt','wb')
>>>a2.write("Python is fun \n Monty python is funnier")
>>>a2.close()
>>>a2.closed
True
>>>a2.write("Add this line")
ValueError                                Traceback (most recent
    call last)
<ipython-input-41-83de982ee0fc> in <module>()
----> 1 a2.write("Add this line")

ValueError: I/O operation on closed file
```

A new file named `fun.txt` is created in the working directory, which is opened in writable binary formatted file. In this file a two lines namely "Python is fun" and "Monty python is funnier" are printed. Then the file is closed and the status is checked that the file is indeed closed. When the file has been close, using `write()` method will call for error, explicitly mentioning that *input/output operations cannot be performed on a closed file.*

`read()` method reads a string from a file. Python strings can have binary data apart from text data.

7.6 Buffering

This feature is incorporated when physical writes are performed on storage devices. For small files, it is not needed. When very big files are dealt with and data speed is slower then buffering (holding the data before recording) is needed and a space in memory is required to hold the data before passing it on to programs for reading and writing. When the value `0` is used to force unbuffered operation, all file write operations are performed immediately. When the value `1` is used, **line buffering** is enforced i.e. output lines are written whenever a line terminator like `\n` is written. Any other positive value for this argument will make a buffer size of that particular value, available for the file operation.

7.7 Summary

File I/O is central part of a python programmer in all domains of studies. Hence present chapter introduced the fundamentals of file I/O operations using python. Various modes of file operations were illustrated and importance of creating, opening and closing files in a particular mode of operations was illustrated. These actions are critical parts of real-world data analysis where input data is usually in terms of files.

8 Functions and Loops

8.1 Introduction

A function is a part of a computer program where a number of programming statements are clubbed as a block. It can be called as and when desired. his enables modular approach to programming tasks and has become most popular amongst programmers now-a-days as modules can be edited with ease instead of finding edition in one big program. Functions receive **input parameters** and **returns** output parameters. Which using a function, **function name** is called along with values for input parameters and after execution, a set of output parameters are returned. Python functions can be defined at any place in the program, irrespective of the place from where they are being called. They can even be defined as a separate file individually or in a combined manner. Also, they can be called any number of times or may not be called too, as per requirements.

8.2 Defining functions

Just like any other language, python has its own way of defining functions. The structure of a Python function is:

```python
def function_name(parameter_1, parameter_2, ...):
  """Descriptive string"""
  # Comment about statements below
  statements
  return return_parameters
```

As seen above, a python function consists of three parts:

1. **Header:** Begins with a keyword (`def`) and ends with a colon (`:`)
2. **Descriptive String**: A string which describes the purpose of character and can be accessed using `help()` function.
3. **Body:** An indented (4 white spaces in general) set of python statements below the header

8.2.1 Function name

Function names follow the same rules as variable names in python. It is a wiser option to give functions a name which is relevant to its description and also to keep it short too.

8.2.2 Descriptive string

An essential part of a defining a function is to define its inner working details as a string. When `help()` is used, this descriptive string is displayed to user to understand the function and its usage. Usage of a descriptive string is not compulsory feature but it is recommended as a good programming practice. Since the description should be as detailed as possible, hence it constitutes a multi-line string. Multi-line strings can be written under triple quotes. Even if description is one line long, it might need to print single or double quotes for emphasizing a word or phrase, hence usage of triple quotes has been mandated in definition of descriptive string.

8.2.3 Indented block of statements

To define the block of statements which are part of a function, indentation is used as a marker. Statements which are indented after first statement to define a function, are part of the body of statement which the function will execute. When a statement is written without indentation, function is exited.

Rules of defining a function name are same that for defining variable names. It is wise to name them with functionally relevant names so that they are easier to be recalled when in need. When `return_parameetrs` are omitted, the functions returns a null object.

For example `fn-hello.py` as shown below, does not take any input parameters and simply executes the statements of printing the string `Hello world`

```
def greet():
    print "Hello world"

greet()
```

fn–hello.py

Running the file `fn-hello.py` produces the output as shown below:

```
>>>python fn-hello.py
>>>Hello World
```

8.2.4 return statement

When a function returns a parameters, it need not always print the same. Returning can be many other kinds of actions apart from simply printing on a screen. Returning can include passing the variables or their values to another function and/or variables, generating a file of code/data, generating graphs and/or storing it as a file in a graphic format as desired etc. Python program `fn-sq.py` prints the square of first 10 integers.

```
def square(x):
    return x*x

for i in range(10):
    squared_i= square(i)
    print i, squared_i
```

fn–sq.py

Result of running the program is given below:

```
>>>0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
```

Here the function `square` is called inside a "for loop" which increments value of variable `i` from 1 to 10 (generated by the in-built function `range()`).

8.3 Multi-input multi-output functions

A function can input and return any number of parameters as shown in python program `def-multi.py`

```python
def sum(a,b):
    c = a+b
    return a,b,c

results = sum(100,102)
print results
```

def–multi.py

The execution results in:

```
>>>(100, 102, 202)
```

8.4 Local and Global variables

8.5 Concept of loops

The main advantage of using computers for calculations are performing repetitive tasks because they can compute faster than humans. The term *loop* is associated with a repetitive calculations because one defined variable names and repetitively shuffles the variable values in a specified sequence generated by a condition. For example, one might like to find square root of first 10 integers. To perform this calculations, one need to run the function `sqrt()` on a list of first 10 integers (which can be generated by `range()`) function. The list of integers can be stored in an integer and this variable can be put into a loop to perform the operation of finding out square root of each member of the list one-by-one.

8.6 for loop

When same operation has to be carried out on given set of data, `for` loop is a good choice. For example, suppose we simply want to print the individual member of the list, then python code given below (code: `ListMember.py`) can be employed.

```
import numpy as np

a = ['a',1,3.14,'name']

for item in a:
    print "The current item is:",
    print item
```

ListMembers.py

The output is printed on the terminal as:

```
The current item is: a
The current item is: 1
The current item is: 3.14
The current item is: name
```

It is worth noting that most of the programming languages employ a logical statement defining the initialization, condition and increment for running the code. Python programs employ a different strategy where an array is employing a condition and loop simply iterates on each member of the array. This is important for python programs since python being interpretive language, is inherently slower in operation. Spending time in checking a condition each time the loop wishes to take a step, is computationally costly affair. Hence python devise the computation in such a way that once a list/array is formed as per the defined condition, loop can then be run of list/array members. In this way, overall computation time can be reduced.

As an example to understand the usage of `for` loop for numerical computation, suppose one wishes to find the even Pythagorean numbers i.e. even numbers a, b, c such that:

$$a^2 + b^2 = c^2$$

This can be accomplished using `for` loop as given in the python code `pytha.py`. In this python code, user inputs a number denoting the maximum number for which this calculation will be performed. This ensures that calculation has a proper *end* condition (without explicit definition).

```
# Program to generate even Pythagorean numbers
# Pythagorean numbers are those numbers for which pythagorus
    equation stands true

from numpy import sqrt
n = raw_input("Please input a maximum number:") # Asks user to
    input a number
n = int(n)+1 # converts the values stored in variable n to
    integer data type and adds 1 so that computation can be done
    if user feeds 0

# Two loops to define arrays for a and b for which c shall be
    computed
for a in range(1,n):
    for b in range(a,n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square)) # c is converted to an integer
        if ((c_square - c**2) == 0): # if square of a and square
     of b is equal to square of c then the result will be zero
            if (c%2 ==0): # checking if c is an even number
                print a, b, c
```

pytha.py

```
>>>Maximal Number? 20
6 8 10
12 16 20
```

As an exercise, one can write few more lines of code to check if there are any prime number triplets as Pythagorean numbers.

8.7 if-else loop

In the example python code `pytha.py`, `if` statement has already been used. It simply checks a condition, runs the loop if condition results `True` boolean data and exits the loop if condition results `False` boolean data. When multiple conditions needs to be checked in a sequence, `if-else` loops are employed where if condition results `True` boolean data, the statement is executed otherwise next condition is checked and similar operation is performed recursively. This action is performed unless all conditions result in returning the `False` boolean data.

```
# Program to calculate shipping cost based on money spent and
    location

total = int(raw_input('What is the total amount for your online
    shopping?\n'))
area = raw_input('''Type "I" if you are shopping within India...
and "O" if you are shopping outside India\n''')

if area == "I":
    if total <= 500:
        print "Shipping Costs INR 20.00"
    elif total <= 1000:
            print "Shipping Costs INR 100.00"
    elif total <= 1500:
            print "Shipping Costs INR 250.00"
    else:
        print "FREE"

if area == "O":
    if total <= 500:
        print "Shipping Costs INR 75.00"
    elif total <= 1000:
        print "Shipping Costs INR 200.00"
    elif total <= 1500:
        print "Shipping Costs INR 500.00"
    else:
        print "FREE"
```

ifelif.py

```
>>>What is the total amount for your online shopping?
2001

Type "I" if you are shopping within India...
and "O" if you are shopping outside India
I
```

```
FREE

>>>What is the total amount for your online shopping?
300

Type "I" if you are shopping within India...
and "O" if you are shopping outside India
I
Shipping Costs INR 20.00
```

8.8 while loop

A **while** loop has following syntax:

```
while expression:
  statement(s)
```

Note that the `statement()s)` are indented for grouping. The `statement(s)` can be single or multiple actions. The `condition` is a logical expression. The loop iterated until the values of logical expression is `True`. As soon as it becomes `False`, the program control is passed to the next line. `while` loop plays an important role in cases where looping must be skipped if condition is not satisfied since none of satement is executed, if logical expression has `False` value.

The program `while.py` gives an example of code demonstrating working of while loop. Here another modules namely `time` is used to time taken by two lines of codes for thier execution. Writing `help(time)` gives important documentation regarding its usage. The function `time.clock()` returns a floating point number which represents CPU time since the start of process or the time when this fucntion is called first. By substracting the two one gets a number depicting number of seconds taken to execute statements.

```
# Program demonstrating usage of while loop

# Program to count number of steps and time taken for thier
  execution

import time #This module is used for timing lines of codes
import numpy as np
```

```

i = 0 # initializing the counter
while(i<10): # counter condition
    start = time.clock() # defining the variable which stores time.clock(value)
    print "Square root of %d  = %3.2f:"%(i,np.sqrt(i)) # printing the number and its squareroot
    i=i+1 # incrementing the counter
    timing = time.clock() - start # prints time taken to execute two lines of code above
    print "Time taken for execution = %e seconds \n" % timing

print "The end" # signifies exiting the loop after condition is satisfied
```

while.py

The result is shown as:

```
>>>Square root of 0  = 0.00:
Time taken for execution = 0.000158652234404 seconds

Square root of 1  = 1.00:
Time taken for execution = 3.97699673158e-05 seconds

Square root of 2  = 1.41:
Time taken for execution = 4.19081375185e-05 seconds

Square root of 3  = 1.73:
Time taken for execution = 2.77962135442e-05 seconds

Square root of 4  = 2.00:
Time taken for execution = 2.22369708354e-05 seconds

Square root of 5  = 2.24:
Time taken for execution = 2.18093368858e-05 seconds

Square root of 6  = 2.45:
Time taken for execution = 2.13817029362e-05 seconds

Square root of 7  = 2.65:
Time taken for execution = 2.1809336431e-05 seconds

Square root of 8  = 2.83:
Time taken for execution = 2.22369708354e-05 seconds

Square root of 9  = 3.00:
Time taken for execution = 2.43751414928e-05 seconds
```

```

The end
```

Note that the output time might be different for each execution even on same computer since time taken to process a line of code is functional of the state of CPU at that particular moment of time.

8.9 Infinite loops

If the logical expression always outputs the boolean value `True`, then the program never stops. These loops are termed as **infinite loops** since they will take infinite time for execution. One of the simplest example is given in python code `infinite.py`

```
i=1
while i==1:
    print i
print "Good bye"
```

infinite–loop.py

Since the condition remains true always so program will never quite displaying the value of `i`, which is 1. It will never print the last line of code. On a Linux machine, one needs to press `CTRL+C` to interrupt the executing and come back to command line.

8.10 while-else

Within a `while` loop, the statements are executed if condition produces a boolean value `True`. Using `else` statement within this structure allows the user the route the flow of program if condition returns the boolean value `False`.

```
i=0

while i<5:
    print i
    i=i+1
else:
    print "the value execeeds 5"
```

while–else.py

The result is shown below:

```
0
1
2
3
4
the value execeeds 5
```

As soon as incremented value becomes 5, the flow is handles by statements under `else` condition.

8.11 Summary

Functions enable modular structure of programs. Also, controlling the flow of information as well as iterations have become the very basis of computational work in most applications. These two actions are performed by loops. Together, they make python a powerful tool for various applications. Modular structure makes it easier to test and debug. Mastering both skills of writing functions as well as choosing proper loop structure have become key indicators for ranking a programmers performance to solve problems using python codes. Hence the present chapter becomes one of the most important ones for programmers.

9 Numerical Computing formalism

9.1 Introduction

Numerical computation enables us to compute solutions to numerical problems, provided we can frame them into a proper format. This requires certain considerations. For example, if we digitize continuous functions, then we are going to introduce certain errors due to the sampling at a finite frequency. Hence a very accurate result would require very fast sampling rate. In cases when a large data set needs to be computed, it becomes computationally intensive and time consuming task. Also one must understand that the numerical solutions are an approximation at best, compared to analytical solutions. The onus of finding their physical meaning and significance lies on us. The art of discarding solutions which do not have a meaning for real world scenario, is something which a scientist/engineer develops over the years. Also, a computational device is just as intelligent as its operator. The law of GIGO (Garbage-In-Garbage-Out) is followed very strictly in this domain.

In the present chapter, we shall try to understand some of the important steps one must consider to solve a physical problem using numerical computation. Defining a problem in proper term is just the first step. Making the right model and then using the right method to solve (solver) enables to distinguish between a naive and experienced scientist/engineer.

9.2 Physical problems

Everything in our physical world is governed by physical laws. Owing to men and women of science who toiled under difficult circumstances and came up with fine solutions to things happening around us, we obtained mathematical theories for physical laws. To test these mathematical formalisms of physical laws, we use numerical computation. If it yields the same results as that of a real experiment, the validate each other. Numerical simulations can remove the need of doing an experiment altogether provided we have a well tested mathematical formalism. For example, nuclear powers of our times need not test nuclear bombs for real any more. The data about nuclear explosion, which was obtained during real nuclear explosions, enabled scientists to model these physical systems quite accurately, thus eliminating the need to a real testing.

Apart from applications like simulating a real experiment, modeling physical problems are good educational exercises. While modeling, hands-on exercises enables students explore the subject in depth and give a proper meaning of topic under study. Solving numerical problem and visualization of results makes the learning permanent and also ignites the research about flaws in mathematical theory which ultimately leads to new discoveries.

9.3 Defining a model

Modeling means writing equations for a physical system. As the name suggests, an equation is about equating two sides. An equation is written using an = sign where terms on left hand side is equal to term on right hand side. The terms on either sides of equations can be numbers or expressions. For example:

$$3x + 4y + 9z = 10$$

This is an equation having a term $3x + 4y + 9z$ on left hand side (LHS) and a term 10 on right hand side (RHS). Please note that whereas LHS is an algebraic term, RHS is a number.

Expressions are written using functions which is simply a relation between two domains. Like $f(x) = y$ is a relation from y to x using rules of algebra. Mathematics has a rich library of functions using which one can make expressions. Choice of proper functions depend on problem. Some functions describe some situations best. For example, oscillatory behavior can be described in a reasonable manner using trigonometric functions like $sin(x), cos(x)$ etc. Objects moving in straight lines can be described well using linear equations like $y = mx + c$ where x is their present position, m is constant rate of change of x w.r.t y and c is the offset position. Objects moving in a curved fashion can be described by various non-linear functions (where power of dependent variable like x above, is not 1).

In real life we can have situations which can be mixture of these scenarios. Like an object can oscillate and move in curved fashion at the same time. In that case we write an expression using mixture of functions or find new functions which could explain the behavior of object. Verifying the functions is done by finding solutions to equations describing the behavior and matching it with observations taken on object. If they match perfectly, we obtain perfect solutions. In most cases, an exact solutions might be difficult to obtain. In these cases, we get an "approximate" solution. If the errors involved while obtaining an approximate solution are within toleration limits, the models can be acceptable.

As discussed above, physical situations can be analytically solved by writing mathematical expressions in terms of functions involving dependent variables. Simplest problems have simple functions between dependent variables with a single equation. There can be situation where multiple equations are needed to explain a physical behavior. In case of multiple equations being solved, the theory of matrix comes handy.

Suppose equations below define the physical behavior of a system:

$$-x + 3y = 4 \tag{9.1}$$
$$2x - 4y = -3 \tag{9.2}$$

Then this system of two equations can be represented by a matrix equation as follows:

$$\begin{bmatrix} -1 & 3 \\ 2 & -4 \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Now using matrix algebra, values of variables x and y can be found such that they satisfy the equations. Those values are called roots of these equations. These roots are the point in 2-D space (because we had 2 dependent variables) where the system will find stability for that physical problem. In this way, we can predict the behavior of system without actually doing an experiment.

Mathematical concept of differentiation and integration becomes very important where we need to work with dynamic system. When the system is constantly changing the values of dependent variables to produce a scenario, then its important to know the rate of change of these variables. When these variables are independent of each other, we use simple derivatives to define their rate of change. When they are not independent of each other, we use partial derivatives for the same.

For example, Newtons second law of motion says that rate of change of velocity of an object is directly proportional to the force applied on it. Mathematically:

$$F \propto \frac{dy}{dx} \tag{9.3}$$

The proportionality is turned into equality by substituting for a constant of multiplication m such that:

$$F = m \times \frac{dy}{dx} \tag{9.4}$$

If we know values or expressions for F, this equation can be solved analytically and solutions can be found to this equation. But in some cases, the analytical solution may be too difficult to obtain. In those cases, we digitize the system and find a numerical solution.

There are many methods to digitize and numerically solve a given function. Programs to implement a particular method to solve a function numerically, is called a solver. A lot of solvers exist to solve a function. Choice of solver is critical to successfully obtain a solution. For example, equation 9.4 is a differential equation. It is a first order ordinary differential equation. A number of solvers exist to solve it like Euler, Runge-Kutta etc. Choice of particular solver depends on accuracy of its solution, time taken for obtaining a solution and amount of memory used during the process. The latter is important where memory is not an freely expendable commodity like micro-computers with limited memory storage.

The advantage of using python to perform a numerical computation lies in the fact that it has a very rich library of modules to perform various tasks required. The predefined functions has been optimized for speed and accuracy (in some cases, accuracy can be predefined). This enables the user to rapidly prototype the problem instead of concentrating on writing functions to do basic tasks and optimizing them for speed, accuracy and memory usage.

9.4 Python Packages

A number of packages exist to perform numerical computation in a particular scientific domain. The website `https://pypi.python.org/pypi` gives a list of packages. Installing package can be simply attained by writing the command

```
>> pip install <package-name>
```

on the LINUX command line.

9.5 Summary

Almost all branch of science and engineering requires one to perform numerical computation. Python is one of the alternative to do so. Python has a library of optimized functions for general computation. Also it has a variety of packages are present to perform a specialized job. This makes it an ideal choice for prototyping a numerical computation problem efficiently.