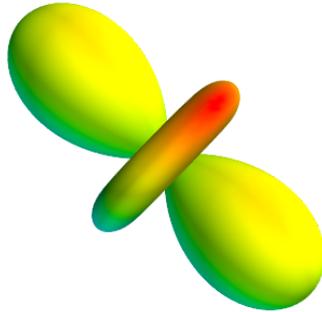# Python for Education

Learning Maths & Science using Python

and

writing them in LaTeX

Ajith Kumar B.P.

Inter University Accelerator Centre

New Delhi 110067

www.iuac.res.in

June 2010

Preface

"Mathematics, rightly viewed, possesses not only truth, but supreme beauty – a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show", wrote Bertrand Russell about the beauty of mathematics. All of us may not reach such higher planes, probably reserved for Russels and Ramanujans, but we also have beautiful curves and nice geometrical figures with intricate symmetries, like fractals, generated by seemingly dull equations. This book attempts to explore it using a simple tool, the Python programming language.

I started using Python for the Phoenix project (www.iuac.res.in). Phoenix was driven in to Python by Pramode CE (pramode.net) and I followed. Writing this document was triggered by some of my friends who are teaching mathematics at Calicut University.

In the first chapter, a general introduction about computers and high level programming languages is given. Basics of Python language, Python modules for array and matrix manipulation, 2D and 3D data visualization, type-setting mathematical equations using latex and numerical methods in Python are covered in the subsequent chapters. Example programs are given for every topic discussed. This document is meant for those who want to try out these examples and modify them for better understanding. Huge amount of material is already available online on the topics covered, and the references to many resources on the Internet are given for the benefit of the serious reader.

This book comes with a live CD, containing a modified version of Ubuntu GNU/Linux operating system. You can boot any PC from this CD and practice Python. Click on the 'Learn by Coding' icon on the desktop to browse through a collection of Python programs, run any of them with a single click. You can practice Python very easily by modifying and running these example programs.

This document is prepared using LyX, a LATEX front-end. It is distributed under the GNU Free Documentation License (www.gnu.org). Feel free to make verbatim copies of this document and distribute through any media. For the LyX source files please contact the author.

Ajith Kumar

IUAC , New Delhi

ajith at iuac.res.in

Contents

Chapter 1

Introduction

Primary objective of this book is to explore the possibilities of using Python language as a tool for learning mathematics and science. The reader is not assumed to be familiar with computer programming. Ability to think logically is enough. Before getting into Python programming, we will briefly explain some basic concepts and tools required.

Computer is essentially an electronic device like a radio or a television. What makes it different from a radio or a TV is its ability to perform different kinds of tasks using the same electronic and mechanical components. This is achieved by making the electronic circuits flexible enough to work according to a set of instructions. The electronic and mechanical parts of a computer are called the Hardware and the set of instructions is called Software (or computer program). Just by changing the Software, computer can perform vastly different kind of tasks. The instructions are stored in binary format using electronic switches.

1.1 Hardware Components

Central Processing Unit (CPU), Memory and Input/Output units are the main hardware components of a computer. CPU[1] can be called the brain of the computer. It contains a Control Unit and an Arithmetic and Logic Unit, ALU. The control unit brings the instructions stored in the main memory one by one and acts according to it. It also controls the movement of data between memory and input/output units. The ALU can perform arithmetic operations like addition, multiplication and logical operations like comparing two numbers.

Memory stores the instructions and data, that is processed by the CPU. All types of information are stored as binary numbers. The smallest unit of memory is called a binary digit or Bit. It can have a value of zero or one. A group of eight bits are called a Byte. A computer has Main and Secondary types of memory. Before processing, data and instructions are moved into the main memory. Main memory is organized into words of one byte size. CPU can select any memory location by using it's address. Main memory is made of semiconductor switches and is very fast. There are two types of Main Memory. Read Only Memory and Read/Write Memory. Read/Write Memory is also called Random Access Memory. All computers contains some programs in ROM which start running when you switch on the machine. Data and programs to be stored for future use are saved to Secondary memory, mainly devices like Hard disks, floppy disks, CDROM or magnetic tapes.

The Input devices are for feeding the input data into the computer. Keyboard is the most common input device. Mouse, scanner etc. are other input devices. The processed data is displayed or printed using the output devices. The monitor screen and printer are the most common output devices.

1.2 Software components

An ordinary user expects an easy and comfortable interaction with a computer, and most of them are least inclined to learn even the basic concepts. To use modern computers for common applications like browsing and word processing, all you need to do is to click on some icons and type on the keyboard. However, to write your own computer programs, you need to learn some basic concepts, like the operating system, editors, compilers, different types of user interfaces etc. This section describes the basics from that point of view.

1.2.1 The Operating System

Operating system (OS) is the software that interacts with the user and makes the hardware resources available to the user. It starts running when you switch on the computer and remains in control. On user request, operating system loads other application programs from disk to the main memory and executes them. OS also provides a file system, a facility to store information on devices like floppy disk and hard disk. In fact the OS is responsible for managing all the hardware resources.

---

[1] The cabinet that encloses most of the hardware is called CPU by some, mainly the computer vendors. They are not referring to the actual CPU chip.

Figure 1.1: A GNU/Linux Terminal.

GNU/Linux and MS Windows are two popular operating systems. Based on certain features, operating systems can be classified as:

- Single user, single process systems like MS DOS. Only one process can run at a time. Such operating systems do not have much control over the application programs.

- Multi-tasking systems like MS Windows, where more than one processe can run at a time.

- Multi-user, multi-tasking systems like GNU/Linux, Unix etc. More than one person can use the computer at the same time.

- Real-time systems, mostly used in control applications, where the response time to any external input is maintained under specified limits.

1.2.2 The User Interface

Interacting with a computer involves, starting various application programs and managing them on the computer screen. The software that manages these actions is called the user interface. The two most common forms of user interface have historically been the Command-line Interface, where computer commands are typed out line-by-line, and the Graphical User Interface (GUI), where a visual environment (consisting of windows, menus, buttons, icons, etc.) is present.

1.2.2.1 The Command Terminal

To run any particular program, we need to request the operating system to do so. Under a Graphical User Interface, we do this by choosing the desired application from a menu. It is possible only because someone has added it to the menu earlier. When you start writing your own programs, obviously they will not appear in any menu. Another way to request the operating system to execute a program is to enter the name of the program (more precisely, the name of the file containing it) at the Command Terminal. On an Ubuntu GNU/Linux system, you can start a Terminal from the menu names Applications->Accessories->Terminal. Figure 1.1 shows a Terminal displaying the list of files in a directory (output of the command 'ls -l' , the -l option is for long listing).

The command processor offers a host of features to make the interaction more comfortable. It keeps track of the history of commands and we can recall previous commands, modify and reuse them using the cursor keys. There is also a completion feature implemented using the Tab key that reduces typing. Use the tab key to complete command and filenames. To run *hello.py* from our test directory, type *python h* and then press the tab key to complete it. If there are more than one file starting with 'h', you need to type more characters until the ambiguity is removed. Always use the up-cursor key to recall the previous commands and re-issue it.

The commands given at the terminal are processed by a program called the *shell*. (The version now popular under GNU/Linux is called bash, the Bourne again shell). Some of the GNU/Linux commands are listed below.

- top : Shows the CPU and memory usage of all the processes started.

- cp filename filename : copies a file to another.
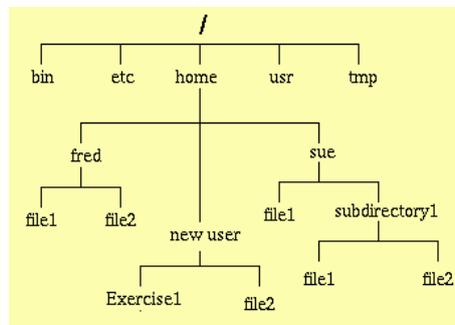
Figure 1.2: The GNU/Linux file system tree.

- mv : moves files from one folder to another, or rename a file.

- rm : deletes files or directories.

- man : display manual pages for a program. For example 'man bash' will give help on the bash shell. Press 'q' to come out of the help screen.

- info : A menu driven information system on various topics.

See the manual pages of 'mv', cp, 'rm' etc. to know more about them. Most of these commands are application programs, stored inside the folders /bin or /sbin, that the shell starts for you and displays their output inside the terminal window.

1.2.3 The File-system

Before the advent of computers, people used to keep documents in files and folders. The designers of the Operating System have implemented the electronic counterparts of the same. The storage space is made to appear as files arranged inside folders (directory is another term for folder). A simplified schematic of the GNU/Linux file system is shown in figure 1.2. The outermost directory is called 'root' directory and represented using the forward slash character. Inside that we have folders named bin, usr, home, tmp etc., containing different type of files.

1.2.3.1 Ownership & permissions

On a multi-user operating system, application programs and document files must be protected against any misuse. This is achieved by defining a scheme of ownerships and permissions. Each and every file on the system will be owned by a specific user. The read, write and execute permissions can be assigned to them, to control the usage. The concept of *group* is introduced to share files between a selected group of users.

There is one special user named *root* (also called the system administrator or the super user) , who has permission to use all the resources. Ordinary user accounts, with a username and password, are created for everyone who wants to use the computer. In a multi-user operating system, like GNU/Linux, every user will have one directory inside which he can create sub-directories and files. This is called the 'home directory' of that user. Home directory of one user cannot be modified by another user.

The operating system files are owned by *root*. The /home directory contains subdirectories named after every ordinary user, for example, the user *fred* owns the directory */home/fred* (fig 1.2) and its contents. That is also called the user's home directory. Every file and directory has three types of permissions : read, write and execute. To view them use the 'ls -l ' command. The first character of output line tells the type of the file. The next three characters show the *rwx* (read, write, execute) permissions for the owner of that file. Next three for the users belonging to the same group and the next three for other users. A hyphen character (-) means the permission corresponding to that field is not granted. For example, the figure 1.1 shows a listing of five files:

1. asecret.dat : read & write for the owner. No one else can even see it.

2. foo.png : rw for owner, but others can view the file.

3. hello.py : rwx for owner, others can view and execute.

4. share.tex : rw for owner and other members of the same group.

5. xdata is a directory. Note that execute permission is required to view contents of a directory.

The system of ownerships and permissions also protects the system from virus attacks[2]. The virus programs damage the system by modifying some application program. On a true multi-user system, for example GNU/Linux, the application program and other system files are owned by the root user and ordinary users have no permission to modify them. When a virus attempts to modify an application, it fails due to this permission and ownership scheme.

1.2.3.2   Current Directory

There is a working directory for every user. You can create subdirectories inside that and change your current working directory to any of them. While using the command-line interface, you can use the 'cd' command to change the current working directory. Figure 1.1 shows how to change the directory and come back to the parent directory by using double dots. We also used the command 'pwd' to print the name of the current working directory.

1.3   Text Editors

To create and modify files, we use different application programs depending on the type of document contained in that file. Text editors are used for creating and modifying plain text matter, without any formatting information embedded inside. Computer programs are plain text files and to write computer programs, we need a text editor. *'gedit'* is a simple, easy to use text editor available on GNU/Linux, which provides syntax high-lighting for several programming languages.

1.4   High Level Languages

In order to solve a problem using a computer, it is necessary to evolve a detailed and precise step by step method of solution. A set of these precise and unambiguous steps is called an Algorithm. It should begin with steps accepting input data and should have steps which gives output data. For implementing any algorithm on a computer, each of it's steps must be converted into proper machine language instructions. Doing this process manually is called Machine Language Programming. Writing machine language programs need great care and a deep understanding about the internal structure of the computer hardware. High level languages are designed to overcome these difficulties. Using them one can create a program without knowing much about the computer hardware.

We already learned that to solve a problem we require an algorithm and it has to be executed step by step. It is possible to express the algorithm using a set of precise and unambiguous notations. The notations selected must be suitable for the problems to be solved. *A high level programming language is a set of well defined notations which is capable of expressing algorithms.*

In general a high level language should have the following features.

1. Ability to represent different data types like characters, integers and real numbers. In addition to this it should also support a collection of similar objects like character strings, arrays etc.

2. Arithmetic and Logical operators that acts on the supported data types.

3. Control flow structures for decision making, branching, looping etc.

4. A set of syntax rules that precisely specify the combination of words and symbols permissible in the language.

5. A set of semantic rules that assigns a single, precise and unambiguous meaning to each syntactically correct statement.

Program text written in a high level language is often called the Source Code. It is then translated into the machine language by using translator programs. There are two types of translator programs, the Interpreter and the Compiler. *Interpreter reads the high level language program line by line, translates and executes it. Compilers convert the entire program in to machine language and stores it to a file which can be executed.*

High level languages make the programming job easier. We can write programs that are machine independent. For the same program different compilers can produce machine language code to run on different types of computers and operating systems. BASIC, COBOL, FORTRAN, C, C++, Python etc. are some of the popular high level languages, each of them having advantages in different fields.

To write any useful program for solving a problem, one has to develop an algorithm. The algorithm can be expressed in any suitable high level language. *Learning how to develop an algorithm is different from learning a programming language. Learning a programming language means learning the notations, syntax and semantic*

---

[2]Do not expect this from the MS Windows system. Even though it allows to create users, any user (by running programs or viruses) is allowed to modify the system files. This may be because it grew from a single process system like MSDOS and still keeps that legacy.

rules of that language. Best way to do this is by writing small programs with very simple algorithms. After becoming familiar with the notations and rules of the language one can start writing programs to implement more complicated algorithms.

1.5 On Free Software

Software that can be used, studied, modified and redistributed in modified or unmodified form without restriction is called Free Software. In practice, for software to be distributed as free software, the human-readable form of the program (the source code) must be made available to the recipient along with a notice granting the above permissions.

The free software movement was conceived in 1983 by Richard Stallman to give the benefit of "software freedom" to computer users. Stallman founded the Free Software Foundation in 1985 to provide the organizational structure to advance his Free Software ideas. Later on, alternative movements like Open Source Software came.

Software for almost all applications is currently available under the pool of Free Software. GNU/Linux operating system, OpenOffice.org office suite, LaTeX typesetting system, Apache web server, GIMP image editor, GNU compiler collection, Python interpreter etc. are some of the popular examples. For more information refer to www.gnu.org website.

1.6 Exercises

1. What are the basic hardware components of a computer.

2. Name the working directory of a user named 'ramu' under GNU/Linux.

3. What is the command to list the file names inside a directory (folder).

4. What is the command under GNU/Linux to create a new folder.

5. What is the command to change the working directory.

6. Can we install more than one operating systems on a single hard disk.

7. Name two most popular Desktop Environments for GNU/Linux.

8. How to open a command window from the main menu of Ubuntu GNU/Linux.

9. Explain the file ownership and permission scheme of GNU/Linux.

Chapter 2

Programming in Python

Python is a simple, high level language with a clean syntax. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code. To know more visit the Python website.[1]

2.1 Getting started with Python

To start programming in Python, we have to learn how to type the source code and save it to a file, using a text editor program. We also need to know how to open a Command Terminal and start the Python Interpreter. The details of this process may vary from one system to another. On an Ubuntu GNU/Linux system, you can open the *Text Editor* and the *Terminal* from the Applications->Accessories menu.

2.1.1 Two modes of using Python Interpreter

If you issue the command 'python', without any argument, from the command terminal, the Python interpreter will start and display a '>>>' prompt where you can type Python statements. Use this method only for viewing the results of single Python statements, for example to use Python as a calculator. It could be confusing when you start writing larger programs, having looping and conditional statements. The preferred way is to enter your source code in a text editor, save it to a file (with .py extension) and execute it from the command terminal using Python. A screen-shot of the Desktop with Text Editor and Terminal is shown in figure 2.1.

In this document, we will start writing small programs showing the essential elements of the language without going into the details. The reader is expected to run these example programs and also to modify them in different ways.It is like learning to drive a car, you master it by practicing.

Let us start with a program to display the words *Hello World* on the computer screen. This is the customary 'hello world' program. There is another version that prints 'Goodbye cruel world', probably invented by those who give up at this point. The Python 'hello world' program is shown below.

Example. hello.py

```
print 'Hello World'
```

This should be entered into a text file using any text editor. On a GNU/Linux system you may use the text editor like 'gedit' to create the source file, save it as *hello.py* . The next step is to call the Python Interpreter to execute the new program. For that, open a command terminal and (at the $ prompt) type: [2]

$ python hello.py

2.2 Variables and Data Types

As mentioned earlier, any high level programming language should support several data types. The problem to be solved is represented using variables belonging to the supported data types. Python supports numeric data types like integers, floating point numbers and complex numbers. To handle character strings, it uses the String data type. Python also supports other compound data types like lists, tuples, dictionaries etc.

---

[1] http://www.python.org/
http://docs.python.org/tutorial/
This document, example programs and a GUI program to browse through them are at
http://www.iuac.res.in/phoenix

[2] For quick practicing, boot from the CD provided with this book and click on the learn-by-coding icon to browse through the example programs given in this book. The browser allows you to run any of them with a single click, modify and save the modified versions.
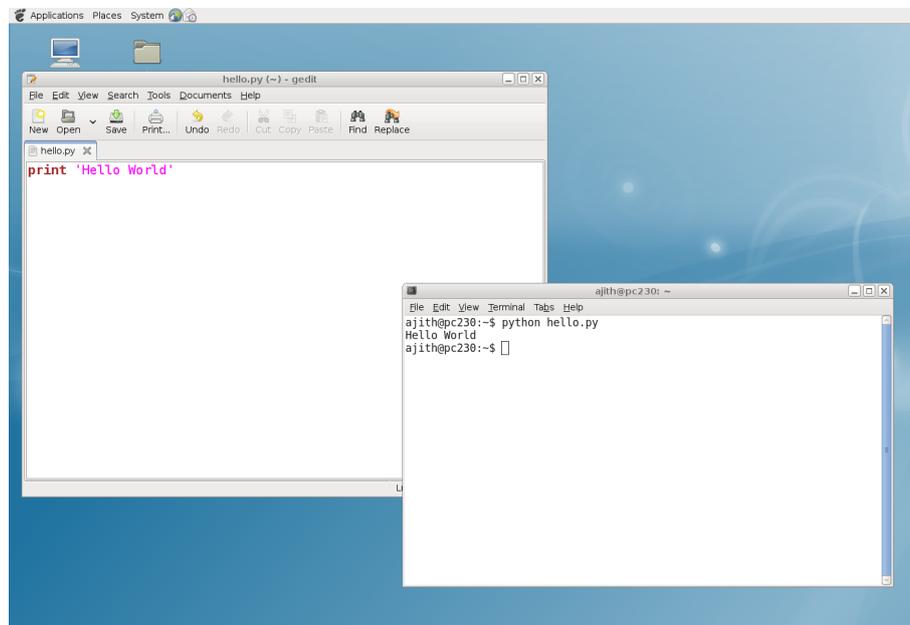
Figure 2.1: Text Editor and Terminal Windows.

In languages like C, C++ and Java, we need to explicitly declare the type of a variable. This is not required in Python. The data type of a variable is decided by the value assigned to it. This is called dynamic data typing. The type of a particular variable can change during the execution of the program. If required, one type of variable can be converted in to another type by explicit type casting, like $y = float(3)$. Strings are enclosed within single quotes or double quotes.

The program *first.py* shows how to define variables of different data types. It also shows how to embed comments inside a program.

Example: first.py

```
'''
A multi-line comment, within a pair of three single quotes.
In a line, anything after a # sign is also a comment
'''
x = 10
print x, type(x)          # print x and its type
x = 10.4
print x, type(x)
x = 3 + 4j
print x, type(x)
x = 'I am a String '
print x,   type(x)
```

The output of the program is shown below. Note that the type of the variable x changes during the execution of the program, depending on the value assigned to it.

```
10 <type 'int'>
10.4 <type 'float'>
(3+4j) <type 'complex'>
I am a String  <type 'str'>
```

The program treats the variables like humans treat labelled envelopes. We can pick an envelope, write some name on it and keep something inside it for future use. In a similar manner the program creates a variable, gives it a name and keeps some value inside it, to be used in subsequent steps. So far we have used four data types of Python: int, float, complex and str. To become familiar with them, you may write simple programs performing arithmetic and logical operations using them.

Example: oper.py

```
x = 2
y = 4
print x + y * 2
```

Operator	Description	Expression	Result
or	Boolean OR	0 or 4	4
and	Boolean AND	3 and 0	0
not x	Boolean NOT	not 0	True
in, not in	Membership tests	3 in [2.2,3,12]	True
<, <=, >, >=, !=, ==	Comparisons	2 > 3	False
|	Bitwise OR	1 | 2	3
^	Bitwise XOR	1 ^ 5	4
&	Bitwise AND	1 & 3	1
<<, >>	Bitwise Shifting	1 << 3	8
+ , -	Add, Subtract	6 - 4	2
*, /, %	Multiply, divide, reminder	5 % 2	1
+x , -x	Positive, Negative	-5*2	-10
~	Bitwise NOT	~1	-2
**	Exponentiation	2 ** 3	8
x[index]	Subscription	a='abcd' ; a[1]	'b'

Table 2.1: Operators in Python listed according to their precedence.

```
s = 'Hello '
print s + s
print 3 * s
print x == y
print y == 2 * x
```

Running the program *oper.py* will generate the following output.

```
10
Hello Hello
Hello Hello Hello
False
True
```

Note that a String can be added to another string and it can be multiplied by an integer. Try to understand the logic behind that and also try adding a String to an Integer to see what is the error message you will get. We have used the logical operator == for comparing two variables.

2.3   Operators and their Precedence

Python supports a large number of arithmetic and logical operators. They are summarized in the table 2.1. An important thing to remember is their precedence. In the expression *2+3*4*, is the addition done first or the multiplication? According to elementary arithmetics, the multiplication should be done first. It means that the multiplication operator has higher precedence than the addition operator. If you want the addition to be done first, enforce it by using parenthesis like $(2+3)*4$. Whenever there is ambiguity in evaluation, use parenthesis to clarify the order of evaluation.

2.4   Python Strings

So far we have come across four data types: Integer, Float, Complex and String. Out of which, String is somewhat different from the other three. It is a collection of same kind of elements, characters. The individual elements of a String can be accessed by indexing as shown in *string.py*. String is a compound, or collection, data type.

Example: string.py

```
s = 'hello world'
print s[0]            # print first element, h
print s[1]            # print e
print s[-1]           # will print the last character
```

Addition and multiplication is defined for Strings, as demonstrated by string2.py.

Example: string2.py

```
a = 'hello'+'world'
print a
b = 'ha' * 3
print b
print a[-1] + b[0]
```

will give the output

helloworld
hahaha
dh

The last element of a and first element of b are added, resulting in the string 'dh'

2.4.1 Slicing

Part of a String can be extracted using the slicing operation. It can be considered as a modified form of indexing a single character. Indexing using $s[a:b]$ extracts elements $s[a]$ to $s[b-1]$. We can skip one of the indices. If the index on the left side of the colon is skipped, slicing starts from the first element and if the index on right side is skipped, slicing ends with the last element.

Example: slice.py

```
a = 'hello world'
print a[3:5]
print a[6:]
print a[:5]
```

The reader can guess the nature of slicing operation from the output of this code, shown below.

```
'lo'
'world'
'hello'
```

Please note that specifying a right side index more than the length of the string is equivalent to skipping it. Modify *slice.py* to print the result of $a[6:20]$ to demonstrate it.

2.5 Python Lists

List is an important data type of Python. It is much more flexible than String. The individual elements can be of any type, even another list. Lists are defined by enclosing the elements inside a pair of square brackets, separated by commas. The program *list1.py* defines a list and print its elements.

Example: list1.py

```
a = [2.3, 3.5, 234]   # make a list
print a[0]
a[1] = 'haha'         # Change an element
print a
```

The output is shown below [3].
2.3
[2.2999999999999998, 'haha', 234]

Lists can be sliced in a manner similar to that if Strings. List addition and multiplication are demonstrated by the following example. We can also have another list as an element of a list.

Example: list2.py

```
a = [1,2]
print a * 2
print a + [3,4]
b = [10, 20, a]
print b
```

The output of this program is shown below.
[1, 2, 1, 2]
[1, 2, 3, 4]
[10, 20, [1, 2]]

---

[3]The floating point number 2.3 showing as 2.2999999999999998 is interesting. This is the very nature of floting point representation of numbers, nothing to do with Python. With the precision we are using, the error in representing 2.3 is around 2.0e-16. This becomes a concern in operations like inversion of big matrices.

2.6   Mutable and Immutable Types

There is one major difference between String and List types, List is mutable but String is not. *We can change the value of an element in a list, add new elements to it and remove any existing element. This is not possible with String type.* Uncomment the last line of *third.py* and run it to clarify this point.

Example: third.py

```
s = [3, 3.5, 234]   # make a list
s[2] = 'haha'       # Change an element
print s
x = 'myname'        # String type
#x[1] = 2           # uncomment to get ERROR
```

The List data type is very flexible, an element of a list can be another list. We will be using lists extensively in the coming chapters. Tuple is another data type similar to List, except that it is immutable. List is defined inside square brackets, tuple is defined in a similar manner but inside parenthesis, like $(3, 3.5, 234)$.

2.7   Input from the Keyboard

Since most of the programs require some input from the user, let us introduce this feature before proceeding further. There are mainly two functions used for this purpose, *input()* for numeric type data and *raw_input()* for String type data. A message to be displayed can be given as an argument while calling these functions.[4]

Example: kin1.py

```
x = input('Enter an integer ')
y = input('Enter one more ')
print 'The sum is ', x + y
s = raw_input('Enter a String ')
print 'You entered ', s
```

It is also possible to read more than one variable using a single input() statement. *String* type data read using raw_input() may be converted into *integer* or *float* type if they contain only the valid characters. In order to show the effect of conversion explicitly, we multiply the variables by 2 before printing. Multiplying a String by 2 prints it twice. If the String contains any other characters than *0..9, . and e*, the conversion to float will give an error.

Example: kin2.py

```
x,y = input('Enter x and y separated by comma ')
print 'The sum is ', x + y
s = raw_input('Enter a decimal number ')
a = float(s)
print s * 2     # prints string twice
print a * 2     # converted value times 2
```

We have learned about the basic data types of Python and how to get input data from the keyboard. This is enough to try some simple problems and algorithms to solve them.

Example: area.py

```
pi = 3.1416
r = input('Enter Radius ')
a = pi * r ** 2        # A = πr²
print 'Area = ', a
```

The above example calculates the area of a circle. Line three calculates r^2 using the exponentiation operator $**$, and multiply it with π using the multiplication operator $*$. r^2 is evaluated first because $**$ has higher precedence than $*$, otherwise the result would be $(\pi r)^2$.

---

[4]Functions will be introduced later. For the time being, understand that it is an isolated piece of code, called from the main program with some input arguments and returns some output.

2.8 Iteration: while and for loops

If programs can only execute from the first line to the last in that order, as shown in the previous examples, it would be impossible to write any useful program. For example, we need to print the multiplication table of eight. Using our present knowledge, it would look like the following

Example: badtable.py

```
print 1 * 8
print 2 * 8
print 3 * 8
print 4 * 8
print 5 * 8
```

Well, we are stopping here and looking for a better way to do this job.

The solution is to use the *while* loop of Python. The logical expression in front of *while* is evaluated, and if it is True, the body of the while loop (the indented lines below the while statement) is executed. The process is repeated until the condition becomes false. We should have some statement inside the body of the loop that will make this condition false after few iterations. Otherwise the program will run in an infinite loop and you will have to press Control-C to terminate it.

The program *table.py*, defines a variable x and assigns it an initial value of 1. Inside the while loop $x * 8$ is printed and the value of x is incremented. This process will be repeated until the value of x becomes greater than 10.

Example: table.py

```
x = 1
while x <= 10:
    print x * 8
    x = x + 1
```

As per the Python syntax, the while statement ends with a colon and the code inside the *while* loop is indented. Indentation can be done using tab or few spaces. In this example, we have demonstrated a simple algorithm.

2.8.1 Python Syntax, Colon & Indentation

Python was designed to be a highly readable language. It has a relatively uncluttered visual layout, uses English keywords frequently where other languages use punctuation, and has notably fewer syntactic constructions than other popular structured languages.

There are mainly two things to remember about Python syntax: *indentation and colon. Python uses indentation to delimit blocks of code.* Both space characters and tab characters are currently accepted as forms of indentation in Python. Mixing spaces and tabs can create bugs that are hard to find, since the text editor does not show the difference. There should not be any extra white spaces in the beginning of any line.

The line before any indented block must end with a colon character.

2.8.2 Syntax of 'for loops'

Python *for* loops are slightly different from the for loops of other languages. Python *for* loop iterates over a compound data type like a String, List or Tuple. During each iteration, one member of the compound data is assigned to the loop variable. The flexibility of this can be seen from the examples below.

Example: forloop.py

```
a = 'Hello'
for ch in a:  # ch is the loop variable
    print ch
b = ['haha', 3.4, 2345, 3+5j]
for item in b:
    print item
```

which gives the output :
```
H
e
l
l
o
```

haha
3.4
2345
(3+5j)

For constructing for loops that executes a fixed number of times, we can create a list using the range() function and run the for loop over that.

Example: forloop2.py

```
mylist = range(5)
print mylist
for item in mylist:
    print item
```

The output will look like :

[0, 1, 2, 3, 4]
0
1
2
3
4

The range function in the above example generates the list $[0, 1, 2, 3, 4]$ and the for loop walks thorugh it printing each member. It is possible to specify the starting point and increment as arguments in the form range(start, end+1, step). The following example prints the table of 5 using this feature.

Example: forloop3.py

```
mylist = range(5,51,5)
for item in mylist:
    print item ,
```

The output is shown below.

5 10 15 20 25 30 35 40 45 50

The print statement inserts a newline at the end by default. We can suppress this behaviour by adding a comma character at the end as done in the previous example.

In some cases, we may need to traverse the list to modify some or all of the elements. This can be done by looping over a list of indices generated by the range() function.For example, the program forloop4.py zeros all the elements of the list.

Example: forloop4.py

```
a = [2, 5, 3, 4, 12]
size = len(a)
for k in range(size):
    a[k] = 0
print a
```

2.9   Conditional Execution: if, elif and else

In some cases, we may need to execute some section of the code only if certain conditions are true. Python implements this feature using the *if, elif* and *else* keywords, as shown in the next example. The indentation levels of *if* and the corresponding *elif* and *else* must be kept the same.

Example: compare.py

```
x = raw_input('Enter a string ')
if x == 'hello':
    print 'You typed ', x
```

Example: big.py

```
x = input('Enter a number ')
if x > 10:
    print 'Bigger Number'
elif x < 10:
    print 'Smaller Number'
else:
    print 'Same Number'
```

The statement *x > 10 and x < 15* can be expressed in a short form, like *10 < x < 15*.

The next example uses *while* and *if* keywords in the same program. Note the level of indentation when the if statement comes inside the while loop. Remember that, the *if* statement must be aligned with the corresponding *elif and else*.

Example: big2.py

```
x = 1
while x < 11:
    if x < 5:
        print 'Small ', x
    else:
        print 'Big ', x
    x = x + 1
print 'Done'
```

2.10   Modify loops : break and continue

We can use the *break* statement to terminate a loop, if some condition is met. The *continue* statement is used to skip the rest of the block and go to the beginning again. Both are demonstrated in the program *big3.py* shown below.

Example: big3.py

```
x = 1
while x < 10:
    if x < 3:
        print 'skipping work', x
        x = x + 1
        continue
    print x
    if x == 4:
        print 'Enough of work'
        break
    x = x  + 1
print 'Done'
```

The output of big3.py is listed below.

skipping work 1

skipping work 2

3

4

Enough of work

Done

Now let us write a program to find out the largest positive number entered by the user. The algorithm works in the following manner. To start with, we assume that the largest number is zero. After reading a number, the program checks whether it is bigger than the current value of the largest number. If so the value of the largest number is replaced with the current number. The program terminates when the user enters zero. Modify max.py to work with negative numbers also.

Example: max.py

```
max = 0
while True:      # Infinite loop
    x = input('Enter a number ')
    if x > max:
        max = x
    if x == 0:
        print max
        break
```

2.11   Line joining

Python interpreter processes the code line by line. A program may have a long line of code that may not physically fit in the width of the text editor. In such cases, we can split a logical line of code into more than one physical lines, using backslash characters (\\), in other words multiple physical lines are joined to form a logical line before interpreting it.

```
if 1900 < year < 2100 and 1 <= month <= 12 :
```

can be split like

```
if 1900 < year < 2100 \
    and 1 <= month <= 12 :
```

Do not split in the middle of words except for Strings. A long String can be split as shown below.

```
longname = 'I am so long and will \
not fit in a single line'
print longname
```

2.12   Exercises

We have now covered the minimum essentials of Python; defining variables, performing arithmetic and logical operations on them and the control flow statements. These are sufficient for handling most of the programming tasks. It would be better to get a grip of it before proceeding further, by writing some code.

1. Modify the expression *print 5+3*2* to get a result of 16

2. What will be the output of *print type(4.5)*

3. Print all even numbers upto 30, suffixed by a * if the number is a multiple of 6. (hint: use % operator)

4. Write Python code to remove the last two characters of 'I am a long string' by slicing, without counting the characters. (hint: use negative indexing)

5. s = '012345' . (a) Slice it to remove last two elements (b) remove first two element.

6. a = [1,2,3,4,5]. Use Slicing and multiplication to generate [2,3,4,2,3,4] from it.

7. Compare the results of 5/2, 5.0/2 and 2.0/3.

8. Print the following pattern using a while loop
   ```
   +
   ++
   +++
   ++++
   ```

9. Write a program to read inputs like 8A, 10C etc. and print the integer and alphabet parts separately.

10. Write code to print a number in the binary format (for example 5 will be printed as 101)

11. Write code to print all perfect cubes upto 2000.

12. Write a Python program to print the multiplication table of 5.

13. Write a program to find the volume of a box with sides 3,4 and 5 inches in cm^3(1 inch = 2.54 cm)

14. Write a program to find the percentage of volume occupied by a sphere of diameter r fitted in a cube of side r. Read r from the keyboard.

15. Write a Python program to calculate the area of a circle.

16. Write a program to divide an integer by another without using the / operator. (hint: use - operator)

17. Count the number of times the character 'a' appears in a String read from the keyboard. Keep on prompting for the string until there is no 'a' in the input.

18. Create an integer division machine that will ask the user for two numbers then divide and give the result. The program should give the result in two parts: the whole number result and the remainder. Example: If a user enters 11 / 4, the computer should give the result 2 and remainder 3.

19. Modify the previous program to avoid division by zero error.

20. Create an adding machine that will keep on asking the user for numbers, add them together and show the total after each step. Terminate when user enters a zero.

21. Modify the adding machine to use raw_input() and check for errors like user entering invalid characters.

22. Create a script that will convert Celsius to Fahrenheit. The program should ask the users to enter the temperature in Celsius and should print out the temperature in Fahrenheit, using $f = \frac{9}{5}c + 32$.

23. Write a program to convert Fahrenheit to Celsius.

24. Create a script that uses a variable and will write 20 times "I will not talk in class." Make each sentence on a separate line.

25. Define $2 + 5j$ and $2 - 5j$ as complex numbers , and find their product. Verify the result by defining the real and imaginary parts separately and using the multiplication formula.

26. Write the multiplication table of 12 using while loop.

27. Write the multiplication table of a number, from the user, using for loop.

28. Print the powers of 2 up to 1024 using a for loop. (only two lines of code)

29. Define the list a = [123, 12.4, 'haha', 3.4]
    a) print all members using a for loop
    b) print the float type members (use type() function)
    c) insert a member after 12.4
    d) append more members

30. Make a list containing 10 members using a for loop.

31. Generate multiplication table of 5 with two lines of Python code. (hint: range function)

32. Write a program to find the sum of five numbers read from the keyboard.

33. Write a program to read numbers from the keyboard until their sum exceeds 200. Modify the program to ignore numbers greater than 99.

34. Write a Python function to calculate the GCD of two numbers

35. Write a Python program to find annual compound interest. Get P,N and R from user

2.13   Functions

Large programs need to be divided into small logical units. A function is generally an isolated unit of code that has a name and does a well defined job. A function groups a number of program statements into a unit and gives it a name. This unit can be invoked from other parts of a program. Python allows you to define functions using the *def* keyword. A function may have one or more variables as arguments, which receive their values from the calling program.

In the example shown below, function arguments (a and b) get the values 3 and 4 respectively from the caller. One can specify more than one variables in the return statement, separated by commas. The function will return a tuple containing those variables. Some functions may not have any arguments, but while calling them we need to use an empty parenthesis, otherwise the function will not be invoked. If there is no return statement, a None is returned to the caller.

Example func.py

```
def sum(a,b):    # a trivial function
    return a + b

print sum(3, 4)
```

The function *factorial.py* calls itself recursively. The value of argument is decremented before each call. Try to understand the working of this by inserting print statements inside the function.

Example factor.py

```
def factorial(n): # a recursive function
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print factorial(10)
```

Example fibanocci.py

```
def fib(n): # print Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b
        a, b = b, a+b

print fib(30)
```

Runing *fibanocci.py* will print

1 1 2 3 5 8 13 21

Modify it to replace $a, b = b, a + b$ by two separate assignment statements, if required introduce a third variable.

2.13.1 Scope of variables

The variables defined inside a function are not known outside the function. There could be two variables, one inside and one outside, with the same name. The program *scope.py* demonstrates this feature.

Example scope.py

```
def change(x):
    counter = x

counter = 10
change(5)
print counter
```

The program will print 10 and not 5. The two variables, both named counter, are not related to each other. In some cases, it may be desirable to allow the function to change some external variable. This can be achieved by using the *global* keyword, as shown in *global.py*.

Example global.py

```
def change(x):
    global counter  # use the global variable
    counter = x

counter = 10
change(5)
print counter
```

The program will now print 5. Functions with global variables should be used carefully to avoid inadvertent side effects.

2.13.2 Optional and Named Arguments

Python allows function arguments to have default values; if the function is called without a particular argument, its default value will be taken. Due to this feature, the same function can be called with different number of arguments. The arguments without default values must appear first in the argument list and they cannot be omitted while invoking the function. The following example shows a function named power() that does exponentiation, but the default value of exponent is set to 2.

Example power.py

```
def power(mant, exp = 2.0):
    return mant ** exp

print power(5., 3)
print power(4.)        # prints 16
print power()          # Gives Error
```

Arguments can be specified in any order by using named arguments.

Example named.py

```
def power(mant = 10.0, exp = 2.0):
    return mant ** exp

print power(5., 3)
print power(4.)        # prints 16
print power(exp=3)     # mant gets 10.0, prints 1000
```

2.14 More on Strings and Lists

Before proceeding further, we will explore some of the functions provided for manipulating strings and lists. Python strings can be manipulated in many ways. The following program prints the length of a string, makes an upper case version for printing and prints a help message on the String class.

Example: stringhelp.py

```
s = 'hello world'
print len(s)
print s.upper()
help(str)              # press q to exit help
```

Python is an object oriented language and all variables are objects belonging to various classes. The method upper() (a function belonging to a class is called a method) is invoked using the dot operator. All we need to know at this stage is that there are several methods that can be used for manipulating objects and they can be invoked like: *variable_name.method_name()*.

2.14.1 split and join

Splitting a String will result in a list of smaller strings. If you do not specify the separator, the space character is assumed by default. To demonstrate the working of these functions, few lines of code and its output are listed below.

Example: split.py

```
s = 'I am a long string'
print s.split()
a = 'abc.abc.abc'
aa = a.split('.')
print aa
mm = '+'.join(aa)
print mm
```

The result is shown below

```
['I', 'am', 'a', 'long', 'string']
['abc', 'abc', 'abc']
'abc+abc+abc'
```

The List of strings generated by split is joined using '+' character, resulting in the last line of the output.

2.14.2   Manipulating Lists

Python lists are very flexible, we can append, insert, delete and modify elements of a list. The program *list3.py* demonstrates some of them.

Example: list3.py

```
a = []          # make an empty list
a.append(3)     # Add an element
a.insert(0,2.5) # insert 2.5 as first element
print a, a[0]
print len(a)
```

The output is shown below.
[2.5, 3] 2.5
2

2.14.3   Copying Lists

Lists cannot be copied like numeric data types. The statement $b = a$ will not create a new list b from list a, it just make a reference to a. The following example will clarify this point. To make a duplicate copy of a list, we need to use the *copy* module.

Example: list_copy.py

```
a = [1,2,3,4]
print a
b = a           # b refers to a
print a == b # True
b[0] = 5        # Modifies a[0]
print a
import copy
c = copy.copy(a)
c[1] = 100
print a is c # is False
print a, c
```

The output is shown below.
[1, 2, 3, 4]
True
[5, 2, 3, 4]
False
[5, 2, 3, 4] [5, 100, 3, 4]

2.15   Python Modules and Packages[5]

One of the major advantages of Python is the availability of libraries for various applications like graphics, networking and scientific computation. The standard library distributed with Python itself has a large number of modules: time, random, pickle, system etc. are some of them. The site http://docs.python.org/library/ has the complete reference.

Modules are loaded by using the *import* keyword. Several ways of using *import* is explained below, using the math (containing mathematical functions) module as an example.

2.15.1   Different ways to import

simplest way to use import is shown in *mathsin.py*, where the function is invoked using the form *module_name.function_name()*. In the next example, we use an alias for the module name.

Example mathsin.py

```
import math
print math.sin(0.5) # module_name.method_name
```

---

[5]While giving names to your Python programs, make sure that you are not directly or indirectly importing any Python module having same name. For example, if you create a program named *math.py* and keep it in your working directory, the *import math* statement from any other program started from that directory will try to import your file named *math.py* and give error. If you ever do that by mistake, delete all the files with .pyc extension from your directory.

Example mathsin2.py

```
import math as m    # Give another name for math
print m.sin(0.5)    # Refer by the new name
```

We can also import the functions to behave like local (like the ones within our source file) function, as shown below. The character * is a wild card for importing all the functions.

Example mathlocal.py

```
from math import sin  # sin is imported as local
print sin(0.5)
```

Example mathlocal2.py

```
from math import *   # import everything from math
print sin(0.5)
```

In the third and fourth cases, we need not type the module name every time. But there could be trouble if two modules imported contains a function with same name. In the program *conflict.py*, the sin() from *numpy* is capable of handling a list argument. After importing *math.py*, line 4, the sin function from *math* module replaces the one from *numpy*. The error occurs because the sin() from *math* can accept only a numeric type argument.

Example conflict.py

```
from numpy import *
x = [0.1, 0.2, 0.3]
print sin(x)           # numpy's sin can handle lists
from math import *     # sin of math becomes effective
print sin(x)           # will give ERROR
```

2.15.2   Packages

Packages are used for organizing multiple modules. The module name A.B designates a submodule named B in a package named A. The concept is demonstrated using the packages Numpy[6] and Scipy.

Example submodule.py

```
import numpy
print numpy.random.normal()
import scipy.special
print scipy.special.j0(.1)
```

In this example *random* is a module inside the package *NumPy*. Similarly *special* is a module inside the package *Scipy*. We use both of them in the package.module.function() format. But there is some difference. In the case of Numpy, the random module is loaded by default, importing scipy does not import the module special by default. This behavior can be defined while writing the Package and it is upto the package author.

2.16   File Input/Output

Disk files can be opened using the function named open() that returns a File object. Files can be opened for reading or writing. There are several methods belonging to the File class that can be used for reading and writing data.

Example wfile.py

```
f = open('test.dat' , 'w')
f.write ('This is a test file')
f.close()
```

Above program creates a new file named 'test.dat' (any existing file with the same name will be deleted) and writes a String to it. The following program opens this file for reading the data.

Example rfile.py

---

[6]NumPy will be discusssed later in chapter 3.

```
f = open('test.dat' , 'r')
print f.read()
f.close()
```

Note that the data written/read are character strings. read() function can also be used to read a fixed number of characters, as shown below.

Example rfile2.py

```
f = open('test.dat' , 'r')
print f.read(7)       # get first seven characters
print f.read()        # get the remaining ones
f.close()
```

Now we will examine how to read a text data from a file and convert it into numeric type. First we will create a file with a column of numbers.

Example wfile2.py

```
f = open('data.dat' , 'w')
for k in range(1,4):
        s = '%3d\n' %(k)
        f.write(s)
f.close()
```

The contents of the file created will look like this.

```
1
2
3
```

Now we write a program to read this file, line by line, and convert the string type data to integer type, and print the numbers.[7]

Example rfile3.py

```
f = open('data.dat' , 'r')
while 1: # infinite loop
    s = f.readline()
    if s == '' :    # Empty string means end of file
        break       # terminate the loop
    m = int(s)      # Convert to integer
    print m * 5
f.close()
```

2.16.1   The pickle module

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the read() method only returns Strings, which will have to be converted in to a number explicitly. However, when you want to save and restore data types like lists, dictionaries, or class instances, things get a lot more complicated. Rather than have the users constantly writing and debugging code to save complicated data types, Python provides a standard module called pickle.

Example pickledump.py

```
import pickle
f = open('test.pck' , 'w')
pickle.dump(12.3, f) # write a float type
f.close()
```

Now write another program to read it back from the file and check the data type.

Example pickleload.py

```
import pickle
f = open('test.pck' , 'r')
x = pickle.load(f)
print x , type(x)         # check the type of data read
f.close()
```

---

[7]This will give error if there is a blank line in the data file. This can be corrected by changing the comparison statement to if *len(s) < 1:* , so that the processing stops at a blank line. Modify the code to skip a blank line instead of exiting (hint: use continue).

Conversion	Conversion	Example	Result
d , i	signed Integer	'%6d'%(12)	' 12'
f	floating point decimal	'%6.4f'%(2.0/3)	0.667
e	floating point exponential	'%6.2e'%(2.0/3)	6.67e-01
x	hexadecimal	'%x'%(16)	10
o	octal	'%o'%(8)	10
s	string	'%s'%('abcd')	abcd
0d	modified 'd'	'%05d'%(12)	00012

Table 2.2: Formatted Printing in Python

2.17 Formatted Printing

Formatted printing is done by using a format string followed by the % operator and the values to be printed. If format requires a single argument, values may be a single variable. Otherwise, values must be a tuple (just place them inside parenthesis, separated by commas) with exactly the number of items specified by the format string.

Example: format.py

```
a = 2.0 /3  # 2/3 will print zero
print a
print 'a = %5.3f' %(a) # upto 3 decimal places
```

Data can be printed in various formats. The conversion types are summarized in the following table. There are several flags that can be used to modify the formatting, like justification, filling etc.

The following example shows some of the features available with formatted printing.

Example: format2.py

```
a = 'justify as you like'
print '%30s'%a          # right justified
print '%-30s'%a         # minus sign for left justification
for k in range(1,11):   # A good looking table
    print '5 x %2d = %2d' %(k, k*5)
```

The output of *format2.py* is given below.

```
         justify as you like
justify as you like
5 x  1 =  5
5 x  2 = 10
5 x  3 = 15
5 x  4 = 20
5 x  5 = 25
5 x  6 = 30
5 x  7 = 35
5 x  8 = 40
5 x  9 = 45
5 x 10 = 50
```

2.18 Exception Handling

Errors detected during execution are called exceptions, like divide by zero. If the program does not handle exceptions, the Python Interpreter reports the exception and terminates the program. We will demonstrate handling exceptions using *try* and *except* keywords, in the example except.py.

Example: except.py

```
x = input('Enter a number ')
try:
    print 10.0/x
except:
    print 'Division by zero not allowed'
```

If any exception occurs while running the code inside the try block, the code inside the except block is executed. The following program implements error checking on input using exceptions.

Example: except2.py

```
def get_number():
    while 1:
        try:
            a = raw_input('Enter a number ')
            x = atof(a)
            return x
        except:
            print 'Enter a valid number'

print get_number()
```

2.19   Turtle Graphics

Turtle Graphics have been noted by many psychologists and educators to be a powerful aid in teaching geometry, spatial perception, logic skills, computer programming, and art. The language LOGO was specifically designed to introduce children to programming, using turtle graphics. An abstract drawing device, called the Turtle, is used to make programming attractive for children by concentrating on doing turtle graphics. It has been used with children as young as 3 and has a track record of 30 years of success in education.

We will use the Turtle module of Python to play with Turtle Graphics and practice the logic required for writing computer programs. Using this module, we will move a *Pen* on a two dimensional screen to generate graphical patterns. The Pen can be controlled using functions like forward(distance), backward(distance), right(angle), left(angle) etc.[8]. Run the program turtle1.py to understand the functions. This section is included only for those who want to practice programming in a more interesting manner.

Example turtle1.py

```
from turtle import *
a = Pen()     # Creates a turtle in a window
a.forward(50)
a.left(45)
a.backward(50)
a.right(45)
a.forward(50)
a.circle(10)
a.up()
a.forward(50)
a.down()
a.color('red')
a.right(90)
a.forward(50)
raw_input('Press Enter')
```

Example turtle2.py

```
from turtle import *
a = Pen()
for k in range(4):
  a.forward(50)
  a.left(90)
  a.circle(25)
raw_input()  # Wait for Key press
```

Outputs of the program turtle2.py and turtle3.py are shown in figure 2.2. Try to write more programs like this to generate more complex patterns.

Example turtle3.py

---

[8]http://docs.python.org/library/turtle.html
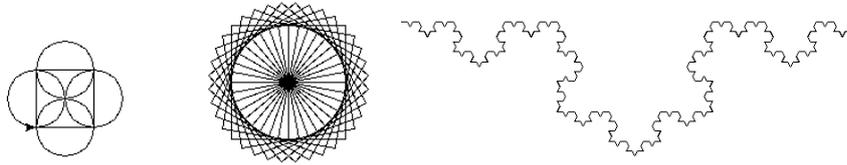
Figure 2.2: Output of turtle2.py (b) turtle3.py (c) turtle4.py

```
from turtle import *
def draw_rectangle():
    for k in range(4):
        a.forward(50)
        a.left(90)

a = Pen()
for k in range(36):
  draw_rectangle()
  a.left(10)
raw_input()
```

The program turtle3.py creates a pattern by drwaing 36 squares, each drawn tilted by 10° from the previous one. The program turtle4.py generates the fractal image as shown in figure2.2(c).

Example turtle4.py

```
from turtle import *

def f(length, depth):
  if depth == 0:
    forward(length)
  else:
    f(length/3, depth-1)
    right(60)
    f(length/3, depth-1)
    left(120)
    f(length/3, depth-1)
    right(60)
    f(length/3, depth-1)

f(500, 4)
raw_input('Press any Key')
```

2.20   Writing GUI Programs

Python has several modules that can be used for creating Graphical User Interfaces. The intention of this chapter is just to show the ease of making GUI in Python and we have selected Tkinter[9], one of the easiest to learn. The GUI programs are event driven (movement of mouse, clicking a mouse button, pressing and releasing a key on the keyboard etc. are called events). The execution sequence of the program is decided by the events, generated mostly by the user. For example, when the user clicks on a Button, the code associated with that Button is executed. GUI Programming is about creating Widgets like Button, Label, Canvas etc. on the screen and executing selected functions in response to events. After creating all the necessary widgets and displaying them on the screen, the control is passed on to Tkinter by calling a function named *mainloop*. After that the program flow is decided by the events and associated callback functions.

For writing GUI programs, the first step is to create a main graphics window by calling the function Tk(). After that we create various Widgets and pack them inside the main window. The example programs given below demonstrate the usage of some of the Tkinter widgets.The program *tkmain.py* is the smallest GUI program one can write using Tkinter. The output of tkmain.py is shown in figure2.3(a).

Example tkmain.py

---

[9]http://www.pythonware.com/library/an-introduction-to-tkinter.htm
http://infohost.nmt.edu/tcc/help/pubs/tkinter/
http://wiki.python.org/moin/TkInter

Figure 2.3: Outputs of (a)tkmain.py (b)tklabel.py
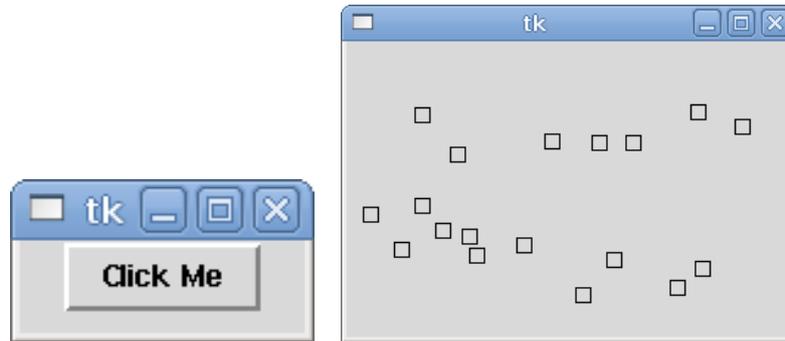
Figure 2.4: Outputs of (a) tkbutton.py (b)tkcanvas.py

```
from Tkinter import *
root = Tk()
root.mainloop()
```

Example tklabel.py

```
from Tkinter import *
root = Tk()
w = Label(root, text="Hello, world")
w.pack()
root.mainloop()
```

The program tklabel.py will generate the output as shown in figure 2.3(b). Terminate the program by clicking on the x displayed at the top right corner. In this example, we used a Label widget to display some text. The next example will show how to use a Button widget.

A Button widget can have a callback function, hello() in this case, that gets executed when the user clicks on the Button. The program will display a Button on the screen. Every time you click on it, the function *hello* will be executed. The output of the program is shown in figure 2.4(a).

Example tkbutton.py

```
from Tkinter import *

def hello():
    print 'hello world'

w = Tk()    # Creates the main Graphics window
b = Button(w, text = 'Click Me', command = hello)
b.pack()
w.mainloop()
```

Canvas is another commonly used widget. Canvas is a drawing area on which we can draw elements like line, arc, rectangle, text etc. The program tkcanvas.py creates a Canvas widget and binds the <Button-1> event to the function draw(). When left mouse button is pressed, a small rectangle are drawn at the cursor position. The output of the program is shown in figure 2.4(b).

Example tkcanvas.py

```
from Tkinter import *

def draw(event):
    c.create_rectangle(event.x, \
            event.y, event.x+5, event.y+5)

w = Tk()
c = Canvas(w, width = 300, height = 200)
c.pack()
c.bind("<Button-1>", draw)
w.mainloop()
```

The next program is a modification of tkcanvas.py. The right mouse-button is bound to remove(). Every time a rectangle is drawn, its return value is added to a list, a global variable, and this list is used for removing the rectangles when right button is pressed.

Example tkcanvas2.py

```
from Tkinter import *
recs = []     # List keeping track of the rectangles

def remove(event):
    global recs
    if len(recs) > 0:
        c.delete(recs[0]) # delete from Canvas
        recs.pop(0)       # delete first item from list

def draw(event):
    global recs
    r = c.create_rectangle(event.x, \
            event.y, event.x+5, event.y+5)
    recs.append(r)

w = Tk()
c = Canvas(w, width = 300, height = 200)
c.pack()
c.bind("<Button-1>", draw)
c.bind("<Button-3>", remove)
w.mainloop()
```

2.21   Object Oriented Programming in Python

OOP is a programming paradigm that uses *objects* (Structures consisting of variables and methods) and their interactions to design computer programs. Python is an object oriented language but it does not force you to make all programs object oriented and there is no advantage in making small programs object oriented. In this section, we will discuss some features of OOP.

Before going to the new concepts, let us recollect some of the things we have learned. We have seen that the effect of operators on different data types is predefined. For example $2*3$ results in 6 and $2*'abc'$ results in $'abcabc'$. This behavior has been decided beforehand, based on some logic, by the language designers. One of the key features of OOP is the ability to create user defined data types. The user will specify, how the new data type will behave under the existing operators like add, subtract etc. and also define methods that will belong to the new data type.

We will design a new data type using the class keyword and define the behavior of it. In the program point.py, we define a class named Point. The variables xpos and ypos are members of Point. The __init__() function is executed whenever we create an instance of this class, the member variables are initialized by this function. The way in which an object belonging to this class is printed is decided by the __str__ function. We also have defined the behavior of add (+) and subtract (-) operators for this class. The + operator returns a new Point by adding the x and y coordinates of two Points. Subtracting a Point from another gives the distance between the two. The method dist() returns the distance of a Point object from the origin. We have not defined the behavior of Point under copy operation. We can use the copy module of Python to copy objects.

Example point.py

```
class Point:
```

```
        '''
        This is documentation comment.
        help(Point) will display this.
        '''
        def __init__(self, x=0, y=0):
          self.xpos = x
          self.ypos = y

        def __str__(self): # overloads print
         return 'Point at (%f,%f)'%(self.xpos, self.ypos)

        def __add__(self, other):  #overloads +
          xpos = self.xpos + other.xpos
          ypos = self.ypos + other.ypos
          return Point(xpos,ypos)

        def __sub__(self, other):  #overloads -
          import math
          dx = self.xpos - other.xpos
          dy = self.ypos - other.ypos
          return math.sqrt(dx**2+dy**2)

        def dist(self):
          import math
          return math.sqrt(self.xpos**2 + self.ypos**2)
```

The program point1.py imports the file point.py to use the class Point defined inside it to demonstrate the properties of the class. A self. is prefixed when a method refers to member of the same object. It refers to the variable used for invoking the method.

Example point1.py

```
    from point import *
    origin = Point()
    print origin
    p1 = Point(4,4)
    p2 = Point(8,7)
    print p1
    print p2
    print p1 + p2
    print p1 - p2
    print p1.dist()
```

Output of program point1.py is shown below.

```
        Point at (0.000000,0.000000)
        Point at (4.000000,4.000000)
        Point at (8.000000,7.000000)
        Point at (12.000000,11.000000)
        5.0
        5.65685424949
```

In this section, we have demonstrated the OO concepts like class, object and operator overloading.

2.21.1   Inheritance, reusing code

Reuse of code is one of the main advantages of object oriented programming. We can define another class that inherits all the properties of the Point class, as shown below. The _ _init_ _ function of colPoint calls the _ _init_ _ function of Point, to get all work except initilization of color done. All other methods and operator overloading defined for Point is inherited by colPoint.

Example cpoint.py

```
    class colPoint(Point):  #colPoint inherits Point
    color = 'black'
    def __init__(self,x=0,y=0,col='black'):
```

```
        Point.__init__(self,x,y)
        self.color = col

    def __str__(self):
        return '%s colored Point at (%f,%f)'% \
            (self.color,self.xpos, self.ypos)
```

Example point2.py

```
from cpoint import *
p1 = Point(5,5)
rp1 = colPoint(2,2,'red')
print p1
print rp1
print rp1 + p1
print rp1.dist()
```

The output of point2.py is listed below.

Point at (5.000000,5.000000)

red colored Point at (2.000000,2.000000)

Point at (7.000000,7.000000)

2.82842712475

For a detailed explanation on the object oriented features of Python, refer to chapters 13, 14 and 15 of the online book http://openbookproject.net//thinkCSpy/

2.21.2   A graphics example program

Object Oriented programming allows us to write Classes with a well defined external interface hiding all the internal details. This example shows a Class named 'disp', for drawing curves, providing the xy coordinates within an arbitrary range . The the world-to-screen coordinate conversion is performed internally. The method named line() accepts a list of xy coordinates. The file *tkplot_ class.py* defines the 'disp' class and is listed below.

Example tkplot_ class.py

```
from Tkinter import *
from math import *
class disp:
    def __init__(self, parent, width=400., height=200.):
        self.parent = parent
        self.SCX = width
        self.SCY = height
        self.border = 1
        self.canvas = Canvas(parent, width=width, height=height)
        self.canvas.pack(side = LEFT)
        self.setWorld(0 , 0, self.SCX, self.SCY)  # scale factors

    def setWorld(self, x1, y1, x2, y2):
        self.xmin = float(x1)
        self.ymin = float(y1)
        self.xmax = float(x2)
        self.ymax = float(y2)
        self.xscale = (self.xmax - self.xmin) / (self.SCX)
        self.yscale = (self.ymax - self.ymin) / (self.SCY)

    def w2s(self, p): #world-to-screen before plotting
        ip = []
        for xy in p:
            ix = self.border + int( (xy[0] - self.xmin) / self.xscale)
            iy = self.border + int( (xy[1] - self.ymin) / self.yscale)
            iy = self.SCY - iy
            ip.append((ix,iy))
        return ip

    def line(self, points, col='blue'):
        ip = self.w2s(points)
        t = self.canvas.create_line(ip, fill=col)
```
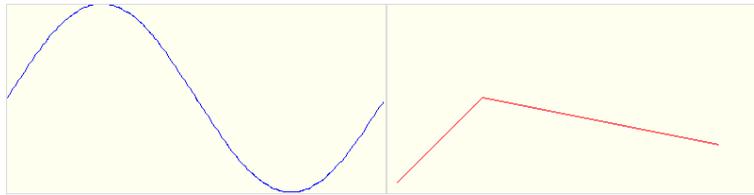
Figure 2.5: Output of tkplot.py

The program *tkplot.py* imports tkplot_class.py and plots two graphs. The advantage of code reuse is evident from this example. [10]. Output of *tkplot.py* is shown in figure 2.5.

Example tkplot.py

```
from tkplot_class import *
from math import *
w = Tk()
gw1 = disp(w)
xy = []
for k in range(200):
    x = 2 * pi * k/200
    y = sin(x)
    xy.append((x,y))
gw1.setWorld(0, -1.0, 2*pi, 1.0)
gw1.line(xy)
gw2 = disp(w)
gw2.line([(10,10),(100,100),(350,50)], 'red')
w.mainloop()
```

2.22   Exercises

1. Generate multiplication table of eight and write it to a file.

2. Make a list and write it to a file using the pickle module.

3. Write a Python program to open a file and write 'hello world' to it.

4. Write a Python program to open a text file and read all lines from it.

5. Write a program to generate the multiplication table of a number from the user. The output should be formatted as shown below
   1 x 5 =   5
   2 x 5 = 10

6. Define the list [1,2,3,4,5,6] using the range function. Write code to insert a 10 after 2, delete 4, add 0 at the end and sort it in the ascending order.

7. Write Python code to generate the sequence of numbers
   25 20 15 10 5
   using range function . Delete 15 from the result and sort it. Print it using a for loop.

8. Define a string *s = 'mary had a little lamb'*.
   a) print it in reverse order
   b) split it using space character as sepatator

9. Join the elements of the list ['I', 'am', 'in', 'pieces'] using + character. Do the same using a for loop also.

10. Create a window with five buttons. Make each button a different color. Each button should have some text on it.

11. Create a program that will put words in alphabetical order. The program should allow the user to enter as many words as he wants to.

12. Create a program that will check a sentence to see if it is a palindrome. A palindrome is a sentence that reads the same backwards and forwards ('malayalam').

---

[10]A more sophisticated version of the *disp* class program (draw.py) is included in the package 'learn-by-coding', available on the CD.

13. A text file contains two columns of numbers. Write a program to read them and print the sum of numbers in each row.

14. Read a String from the keyboard. Multiply it by an integer to make its length more than 50. How do you find out the smallest number that does the job.

15. Write a program to find the length of the hypotenuse of a right triangle from the length of other two sides, get the input from the user.

16. Write a program displaying 2 labels and 2 buttons. It should print two different messages when clicked on the two buttons.

17. Write a program with a Canvas and a circle drawn on it.

18. Write a program using for loop to reverse a string.

19. Write a Python function to calculate the GCD of two numbers

20. Write a program to print the values of sine function from 0 to 2π with 0.1 increments. Find the mean value of them.

21. Generate N random numbers using random.random() and find out howmay are below 0.5 . Repeat the same for different values of N to draw some conclusions.

22. Use the equation $x = (-b \pm \sqrt{b^2 - 4ac})/2a$ to find the roots of $3x^2 + 6x + 12 = 0$

23. Write a program to calculate the distance between points (x1,y1) and (x2,y2) in a Cartesian plane. Get the coordinates from the user.

24. Write a program to evaluate $y = \sqrt{2.3a} + a^2 + 34.5$ for a = 1, 2 and 3.

25. Print Fibanocci numbers upto 100, without using multiple assignment statement.

26. Draw a chess board pattern using turtle graphics.

27. Find the syntax error in the following code and correct it.
    x=1
    while x <= 10:
    print x * 5

Chapter 3

Arrays and Matrices

In the previous chapter, we have learned the essential features of Python language. We also used the *math* module to calculate trigonometric functions. Using the tools introduced so far, let us generate the data points to plot a sine wave. The program sine.py generates the coordinates to plot a sine wave.

Example sine.py

```
import math
x = 0.0
while x < 2 * math.pi:
    print x , math.sin(x)
    x = x + 0.1
```

The output to the screen can be redirected to a file as shown below, from the command prompt. You can plot the data using some program like xmgrace.

$ python sine.py > sine.dat

$ xmgrace sine.dat

It would be better if we could write such programs without using loops explicitly. Serious scientific computing requires manipulating of large data structures like matrices. The *list* data type of Python is very flexible but the performance is not acceptable for large scale computing. The need of special tools is evident even from the simple example shown above. *NumPy* is a package widely used for scientific computing with Python.[1]

3.1 The NumPy Module

The *numpy* module supports operations on compound data types like arrays and matrices. *First thing to learn is how to create arrays and matrices using the numpy package.* Python lists can be converted into multi-dimensional arrays. There are several other functions that can be used for creating matrices. The mathematical functions like sine, cosine etc. of numpy accepts array objects as arguments and return the results as arrays objects. NumPy arrays can be indexed, sliced and copied like Python Lists.

In the examples below, we will import numpy functions as local (using the syntax *from numpy import **). Since it is the only package used there is no possibility of any function name conflicts.

Example numpy1.py

```
from numpy import *
x = array( [1, 2, 3] )   # Make array from list
print x , type(x)
```

In the above example, we have created an array from a list.

3.1.1 Creating Arrays and Matrices

We can also make multi-dimensional arrays. Remember that a member of a list can be another list. The following example shows how to make a two dimensional array.

Example numpy3.py

---

[1] http://numpy.scipy.org/
http://www.scipy.org/Tentative_NumPy_Tutorial
http://www.scipy.org/Numpy_Functions_by_Category
http://www.scipy.org/Numpy_Example_List_With_Doc

```
from numpy import *
a = [ [1,2] , [3,4] ] # make a list of lists
x = array(a)     # and convert to an array
print a
```

Other than than *array(),* there are several other functions that can be used for creating different types of arrays and matrices. Some of them are described below.

3.1.1.1 arange(start, stop, step, dtype = None)

Creates an evenly spaced one-dimensional array. Start, stop, stepsize and datatype are the arguments. If datatype is not given, it is deduced from the other arguments. Note that, the values are generated within the interval, including start but excluding stop.

arange(2.0, 3.0, .1) makes the array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])

3.1.1.2 linspace(start, stop, number of elements)

Similar to arange(). Start, stop and number of samples are the arguments.

linspace(1, 2, 11) is equivalent to array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.])

3.1.1.3 zeros(shape, datatype)

Returns a new array of given shape and type, filled zeros. The arguments are shape and datatype. For example *zeros([3,2], 'float')* generates a 3 x 2 array filled with zeros as shown below. If not specified, the type of elements defaults to int.

0.0   0.0   0.0
0.0   0.0   0.0

3.1.1.4 ones(shape, datatype)

Similar to zeros() except that the values are initialized to 1.

3.1.1.5 random.random(shape)

Similar to the functions above, but the matrix is filled with random numbers ranging from 0 to 1, of *float* type. For example, random.random([3,3]) will generate the 3x3 matrix;

```
array([[ 0.3759652 , 0.58443562, 0.41632997],
       [ 0.88497654, 0.79518478, 0.60402514],
       [ 0.65468458, 0.05818105, 0.55621826]])
```

3.1.1.6 reshape(array, newshape)

We can also make multi-dimensions arrays by reshaping a one-dimensional array. The function *reshape()* changes dimensions of an array. The total number of elements must be preserved. Working of *reshape()* can be understood by looking at *reshape.py* and its result.

Example reshape.py

```
from numpy import *
a = arange(20)
b = reshape(a, [4,5])
print b
```

The result is shown below.

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

The program *numpy2.py* demonstrates most of the functions discussed so far.

Example numpy2.py

```
from numpy import *
a = arange(1.0, 2.0, 0.1)
print a
b = linspace(1,2,11)
print b
c = ones(5,'float')
print c
d = zeros(5, 'int')
print d
e = random.rand(5)
print e
```

Output of this program will look like;

[1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9]
[1. 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.]
[1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0.]
[0.89039193 0.55640332 0.38962117 0.17238343 0.01297415]

3.1.2 Copying

Numpy arrays can be copied using the copy method, as shown below.

Example array_ copy.py

```
from mumpy import *
a = zeros(5)
print a
b = a
c = a.copy()
c[0] = 10
print a, c
b[0] = 10
print a,c
```

The output of the program is shown below. The statement $b = a$ does not make a copy of a. Modifying b affects a, but c is a separate entity.

[0. 0. 0.]
[0. 0. 0.] [10. 0. 0.]
[10. 0. 0.] [10. 0. 0.]

3.1.3 Arithmetic Operations

Arithmetic operations performed on an array is carried out on all individual elements. Adding or multiplying an array object with a number will multiply all the elements by that number. However, adding or multiplying two arrays having identical shapes will result in performing that operation with the corresponding elements. To clarify the idea, have a look at *aroper.py* and its results.

Example aroper.py

```
from numpy import *
a = array([[2,3], [4,5]])
b = array([[1,2], [3,0]])
print a + b
print a * b
```

The output will be as shown below

```
array([[3, 5],
       [7, 5]])
array([[ 2, 6],
       [12, 0]])
```

Modifying this program for more operations is left as an exercise to the reader.

3.1.4 cross product

Returns the cross product of two vectors, defined by

$$A \times B = \begin{vmatrix} i & j & k \\ A_1 & A_2 & A_3 \\ B_1 & B_2 & B_3 \end{vmatrix} = i(A_2B_3 - A_3B_2) + j(A_1B_3 - A_3B_1) + k(A_1B_2 - A_2B_1) \tag{3.1}$$

It can be evaluated using the function cross((array1, array2). The program *cross.py* prints [-3, 6, -3]

Example cross.py

```
from numpy import *
a = array([1,2,3])
b = array([4,5,6])
c = cross(a,b)
print c
```

3.1.5 dot product

Returns the dot product of two vectors defined by $A.B = A_1B_1 + A_2B_2 + A_3B_3$. If you change the fourth line of *cross.py* to $c = dot(a, b)$, the result will be 32.

3.1.6 Saving and Restoring

An array can be saved to text file using *array.tofile(filename)* and it can be read back using *array=fromfile()* methods, as shown by the code fileio.py

Example fileio.py

```
from numpy import *
a = arange(10)
a.tofile('myfile.dat')
b = fromfile('myfile.dat',dtype = 'int')
print b
```

The function fromfile() sets dtype='float' by default. In this case we have saved an integer array and need to specify that while reading the file. We could have saved it as float the the statement a.tofile('myfile.dat', 'float').

3.1.7 Matrix inversion

The function *linalg.inv(matrix)* computes the inverse of a square matrix, if it exists. We can verify the result by multiplying the original matrix with the inverse. Giving a singular matrix as the argument should normally result in an error message. In some cases, you may get a result whose elements are having very high values, and it indicates an error.

Example inv.py

```
from numpy import *
a = array([ [4,1,-2], [2,-3,3], [-6,-2,1] ], dtype='float')
ainv = linalg.inv(a)
print ainv
print dot(a,ainv)
```

Result of this program is printed below.

```
[[ 0.08333333  0.08333333 -0.08333333]
 [-0.55555556 -0.22222222 -0.44444444]
 [-0.61111111  0.05555556 -0.38888889]]
[[ 1.00000000e+00 -1.38777878e-17 0.00000000e+00]
 [-1.11022302e-16  1.00000000e+00 0.00000000e+00]
 [ 0.00000000e+00  2.08166817e-17 1.00000000e+00]]
```

3.2   Vectorized Functions

The functions like sine, log etc. from NumPy are capable of accepting arrays as arguments. This eliminates the need of writing loops in our Python code.

Example vfunc.py

```
from numpy import *
a = array([1,10,100,1000])
print log10(a)
```

The output of the program is [0. 1. 2. 3.] , where the log of each element is calculated and returned in an array. This feature simplifies the programs a lot. Numpy also provides a function to vectorize functions written by the user.

Example vectorize.py

```
from numpy import *

def spf(x):
    return 3*x

vspf = vectorize(spf)
a = array([1,2,3,4])
print vspf(a)
```

The output will be [3 6 9 12] .

3.3   Exercises

1. Write code to make a one dimensional matrix with elements 5,10,15,20 and 25. make another matrix by slicing the first three elements from it.

2. Create a 3×2 matrix and print the sum of its elements using for loops.

3. Create a 2×3 matrix and fill it with random numbers.

4. Use linspace to make an array from 0 to 10, with stepsize of 0.1

5. Use arange to make an 100 element array ranging from 0 to 10

6. Make an array a = [2,3,4,5] and copy it to b. change one element of b and print both.

7. Make a 3x3 matrix and multipy it by 5.

8. Create two 3x3 matrices and add them.

9. Write programs to demonstrate the dot and cross products.

10. Using matrix inversion, solve the system of equations
    $4x1 - 2x2 + x3 = 11$
    $-2x1 + 4x2 - 2x3 = -16$
    $x1 - 2x2 + 4x3 = 17$

11. Find the new values of the coordinate (10,10) under a rotation by angle $\pi/4$.

12. Write a vectorized function to evaluate $y = x^2$ and print the result for x=[1,2,3].

Chapter 4

Data visualization

A graph or chart is used to present numerical data in visual form. A graph is one of the easiest ways to compare numbers. They should be used to make facts clearer and more understandable. Results of mathematical computations are often presented in graphical format. In this chapter, we will explore the Python modules used for generating two and three dimensional graphs of various types.

4.1 The Matplotlib Module

Matplotlib is a python package that produces publication quality figures in a variety of hardcopy formats. It also provides many functions for matrix manipulation. You can generate plots, histograms, power spectra, bar charts, error-charts, scatter-plots, etc, with just a few lines of code and have full control of line styles, font properties, axes properties, etc. The data points to the plotting functions are supplied as Python lists or Numpy arrays.

If you import matplotlib as *pylab*, the plotting functions from the submodules *pyplot* and matrix manipulation functions from the submodule *mlab* will be available as local functions. Pylab also imports Numpy for you. Let us start with some simple plots to become familiar with matplotlib.[1]

Example plot1.py

```
from pylab import *
data = [1,2,5]
plot(data)
show()
```

In the above example, the x-axis of the three points is taken from 0 to 2. We can specify both the axes as shown below.

Example plot2.py

```
from pylab import *
x = [1,2,5]
y = [4,5,6]
plot(x,y)
show()
```

By default, the color is blue and the line style is continuous. This can be changed by an optional argument after the coordinate data, which is the format string that indicates the color and line type of the plot. The default format string is 'b-' (blue, continuous line). Let us rewrite the above example to plot using red circles. We will also set the ranges for x and y axes and label them.

Example plot3.py

```
from pylab import *
x = [1,2,5]
y = [4,5,6]
plot(x,y,'ro')
xlabel('x-axis')
ylabel('y-axis')
axis([0,6,1,7])
show()
```
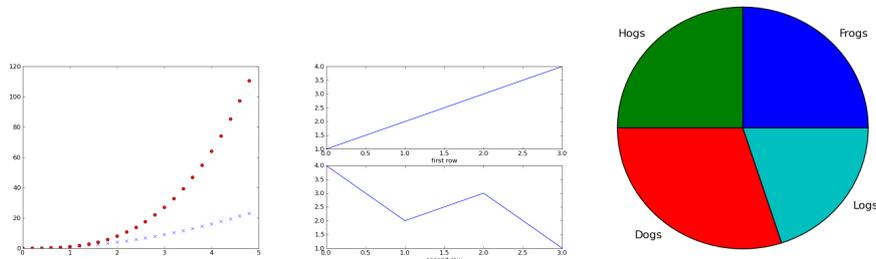
Figure 4.1: Output of (a) plot4.py (b) subplot1.py (c) piechart.py

The figure 4.1 shows two different plots in the same window, using different markers and colors.

Example plot4.py

```
from pylab import *
t = arange(0.0, 5.0, 0.2)
plot(t, t**2,'x')        # t²
plot(t, t**3,'ro')       # t³
show()
```

We have just learned how to draw a simple plot using the pylab interface of matplotlib.

4.1.1   Multiple plots

Matplotlib allows you to have multiple plots in the same window, using the subplot() command as shown in the example subplot1.py, whose output is shown in figure 4.1(b).

Example subplot1.py

```
from pylab import *
subplot(2,1,1)                # the first subplot
plot([1,2,3,4])
subplot(2,1,2)                # the second subplot
plot([4,2,3,1])
show()
```

The arguments to subplot function are NR (number of rows) , NC (number of columns) and a figure number, that ranges from 1 to $NR * NC$. The commas between the arguments are optional if $NR * NC < 10$, ie. subplot(2,1,1) can be written as subplot(211).

Another example of subplot is given is *subplot2.py*. You can modify the variable NR and NC to watch the results. Please note that the % character has different meanings. In *(pn+1)%5*, it is the reminder operator resulting in a number less than 5. The % character also appears in the String formatting.

Example subplot2.py

```
from pylab import *
mark = ['x','o','^','+','>']
NR = 2  # number of rows
NC = 3  # number of columns
pn = 1
for row in range(NR):
    for col in range(NC):
        subplot(NR, NC, pn)
        a = rand(10) * pn
        plot(a, marker = mark[(pn+1)%5])
        xlabel('plot %d X'%pn)
        ylabel('plot %d Y'%pn)
        pn = pn + 1
show()
```

---

[1]http://matplotlib.sourceforge.net/
http://matplotlib.sourceforge.net/users/pyplot_tutorial.html
http://matplotlib.sourceforge.net/examples/index.html
http://matplotlib.sourceforge.net/api/axes_api.html

4.1.2   Polar plots

Polar coordinates locate a point on a plane with one distance and one angle. The distance 'r' is measured from the origin. The angle θ is measured from some agreed starting point. Use the positive part of the $x - axis$ as the starting point for measuring angles. Measure positive angles anti-clockwise from the positive $x - axis$ and negative angles clockwise from it.

Matplotlib supports polar plots, using the $polar(\theta, r)$ function. Let us plot a circle using polar(). For every point on the circle, the value of $radius$ is the same but the polar angle θ changes from 0 to 2π. Both the coordinate arguments must be arrays of equal size. Since θ is having 100 points , r also must have the same number. This array can be generated using the $ones()$ function. The axis($[\theta_{min}, \theta_{max}, r_{min}, r_{max}]$) function can be used for setting the scale.

Example polar.py

```
from pylab import *
th = linspace(0,2*pi,100)
r = 5 * ones(100)  # radius = 5
polar(th,r)
show()
```

4.1.3   Pie Charts

An example of a pie chart is given below. The percentage of different items and their names are given as arguments. The output is shown in figure 4.1(c).

Example piechart.py

```
from pylab import *
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
fracs = [25, 25, 30, 20]
pie(fracs, labels=labels)
show()
```

4.2   Plotting mathematical functions

One of our objectives is to understand different mathematical functions better, by plotting them graphically. We will use the *arange*, *linspace* and *logspace* functions from *numpy* to generate the input data and also the vectorized versions of the mathematical functions. For arange(), the third argument is the stepsize. The total number of elements is calculated from start, stop and stepsize. In the case of linspace(), we provide start, stop and the total number of points. The step size is calculated from these three parameters. Please note that to create a data set ranging from 0 to 1 (including both) with a stepsize of 0.1, we need to specify linspace(0,1,11) and not linspace(0,1,10).

4.2.1   Sine function and friends

Let the first example be the familiar sine function. The input data is from $-\pi$ to $+\pi$ radians[2]. To make it a bit more interesting we are plotting $\sin x^2$ also. The objective is to explain the concept of odd and even functions. Mathematically, we say that a function $f(x)$ is even if $f(x) = f(-x)$ and is odd if $f(-x) = -f(x)$. Even functions are functions for which the left half of the plane looks like the mirror image of the right half of the plane. From the figure 4.2(a) you can see that $\sin x$ is odd and $\sin x^2$ is even.

Example npsin.py

```
from pylab import *
x = linspace(-pi, pi , 200)
y = sin(x)
y1 = sin(x*x)
plot(x,y)
plot(x,y1,'r')
show()
```

Exercise: Modify the program *npsin.py* to plot $\sin^2 x$, $\cos x$, $\sin x^3$ etc.

---

[2]Why do we need to give the angles in radians and not in degrees. Angle in radian is the length of the arc defined by the given angle, with unit radius. Degree is just an arbitrary unit.
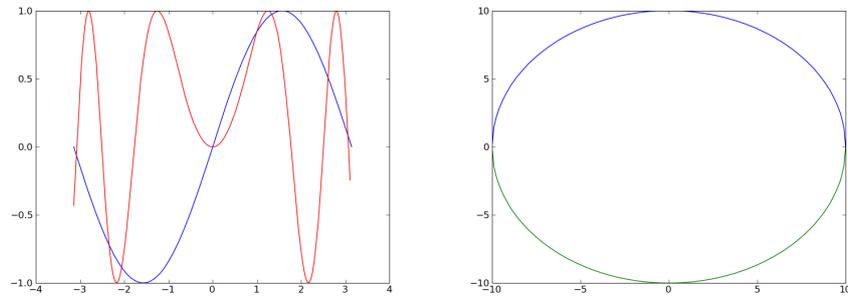
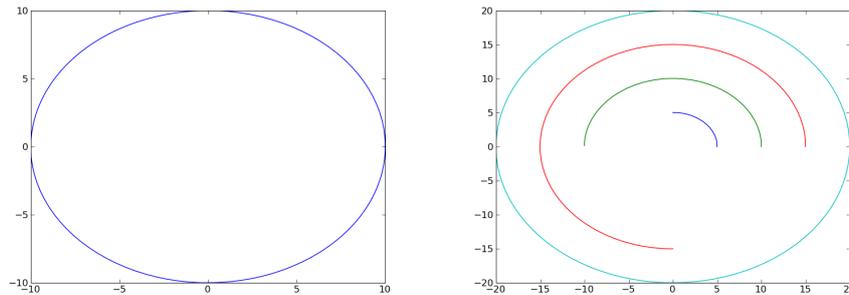Figure 4.2: (a) Output of npsin.py (b) Output of circ.py .

Figure 4.3: (a)Output of circpar.py. (b)Output of arcs.py

4.2.2   Trouble with Circle

Equation of a circle is $x^2 + y^2 = a^2$, where a is the radius and the circle is located at the origin of the coordinate system. In order to plot it using Cartesian coordinates, we need to express y in terms of x, and is given by

$$y = \sqrt{a^2 - x^2}$$

We will create the x-coordinates ranging from $-a$ to $+a$ and calculate the corresponding values of y. This will give us only half of the circle, since for each value of x, there are two values of y (+y and -y). The following program *circ.py* creates both to make the complete circle as shown in figure 4.2(b). Any multi-valued function will have this problem while plotting. Such functions can be plotted better using parametric equations or using the polar plot options, as explained in the coming sections.

Example circ.py

```
from pylab import *
a = 10.0
x = linspace(-a, a , 200)
yupper = sqrt(a**2 - x**2)
ylower = -sqrt(a**2 - x**2)
plot(x,yupper)
plot(x,ylower)
show()
```

4.2.3   Parametric plots

The circle can be represented using the equations $x = a\cos\theta$ and $y = a\sin\theta$. To get the complete circle θ should vary from zero to 2π radians. The output of circpar.py is shown in figure 4.3(a).

Example circpar.py

```
from pylab import *
a = 10.0
th = linspace(0, 2*pi, 200)
x = a * cos(th)
y = a * sin(th)
plot(x,y)
show()
```

Changing the range of θ to less than 2π radians will result in an arc. The following example plots several arcs with different radii. The *for* loop will execute four times with the values of radius 5,10,15 and 20. The range of θ also depends on the loop variable. For the next three values it will be $\pi, 1.5\pi$ and 2π respectively. The output is shown in figure 4.3(b).

Example arcs.py

```
from pylab import *
a = 10.0
for a in range(5,21,5):
    th = linspace(0, pi * a/10, 200)
    x = a * cos(th)
    y = a * sin(th)
    plot(x,y)
show()
```

4.3   Famous Curves

Connection between different branches of mathematics like trigonometry, algebra and geometry can be understood by geometrically representing the equations. You will find a large number of equations generating geometric patterns having interesting symmetries. A collection of them is available on the Internet [2][3]. We will select some of them and plot here. Exploring them further is left as an exercise to the reader.

4.3.1   Astroid

The astroid was first discussed by Johann Bernoulli in 1691-92. It also appears in Leibniz's correspondence of 1715. It is sometimes called the tetracuspid for the obvious reason that it has four cusps. A circle of radius 1/4 rolls around inside a circle of radius 1 and a point on its circumference traces an astroid. The Cartesian equation is

$$x^{\frac{2}{3}} + y^{\frac{2}{3}} = a^{\frac{2}{3}} \tag{4.1}$$

The parametric equations are

$$x = a\cos^3(t), y = a\sin^3(t) \tag{4.2}$$

In order to plot the curve in the Cartesian system, we rewrite equation 4.1 as

$$y = (a^{\frac{2}{3}} - x^{\frac{2}{3}})^{\frac{3}{2}}$$

The program *astro.py* plots the part of the curve in the first quadrant. The program *astropar.py* uses the parametric equation and plots the complete curve. Both are shown in figure 4.4

Example astro.py

```
from pylab import *
a = 2
x = linspace(0,a,100)
y = ( a**(2.0/3) - x**(2.0/3) )**(3.0/2)
plot(x,y)
show()
```

Example astropar.py

```
from pylab import *
a = 2
t = linspace(-2*a,2*a,101)
x = a * cos(t)**3
y = a * sin(t)**3
plot(x,y)
show()
```
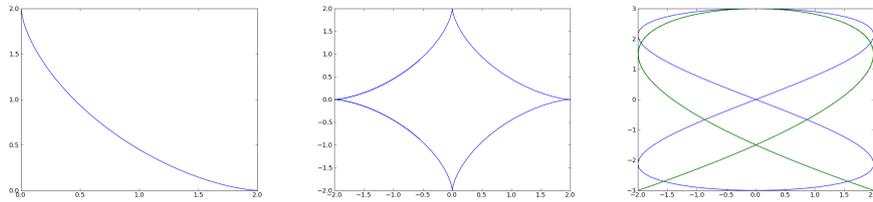
Figure 4.4: (a)Output of astro.py (b) astropar.py (c) lissa.py

4.3.2   Ellipse

The ellipse was first studied by Menaechmus [4] . Euclid wrote about the ellipse and it was given its present name by Apollonius. The focus and directrix of an ellipse were considered by Pappus. Kepler, in 1602, said he believed that the orbit of Mars was oval, then he later discovered that it was an ellipse with the sun at one focus. In fact Kepler introduced the word *focus* and published his discovery in 1609.

The Cartesian equation is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{4.3}$$

The parametric equations are

$$x = a\cos(t), y = b\sin(t) \tag{4.4}$$

The program *ellipse.py* uses the parametric equation to plot the curve. Modifying the parametric equations will result in Lissajous figures. The output of lissa.py are shown in figure 4.4(c).

Example ellipse.py

```
from pylab import *
a = 2
b = 3
t = linspace(0, 2 * pi, 100)
x = a * sin(t)
y = b * cos(t)
plot(x,y)
show()
```

Example lissa.py

```
from pylab import *
a = 2
b = 3
t= linspace(0, 2*pi,100)
x = a * sin(2*t)
y = b * cos(t)
plot(x,y)
x = a * sin(3*t)
y = b * cos(2*t)
plot(x,y)
show()
```

The Lissajous curves are closed if the ratio of the arguments for sine and cosine functions is an integer. Otherwise open curves will result, both are shown in figure 4.4(c).

4.3.3   Spirals of Archimedes and Fermat

The spiral of Archimedes is represented by the equation $r = a\theta$. Fermat's Spiral is given by $r^2 = a^2\theta$. The output of archi.py and fermat.py are shown in figure 4.5.

Example archi.py

```
from pylab import *
a = 2
th= linspace(0, 10*pi,200)
r = a*th
polar(th,r)
axis([0, 2*pi, 0, 70])
show()
```
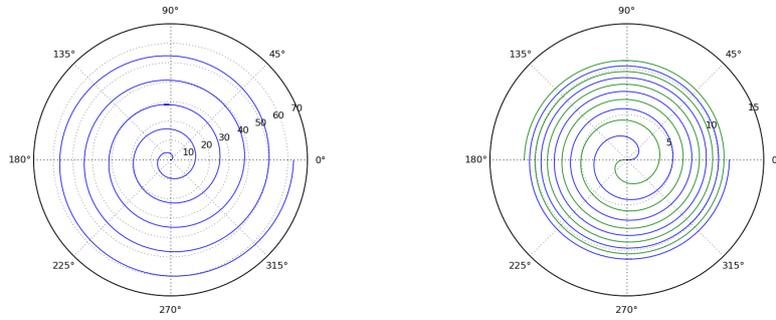
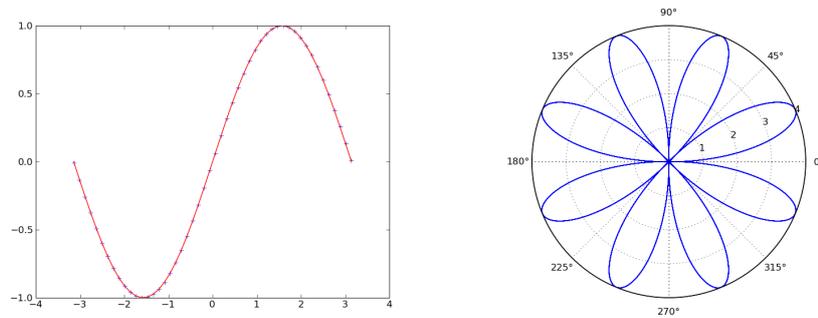Figure 4.5: (a)Archimedes Spiral (b)Fermat's Spiral (c)Polar Rose

Figure 4.6: Outputs of (a)series_sin.py (b) rose.py

Example fermat.py

```
from pylab import *
a = 2
th= linspace(0, 10*pi,200)
r = sqrt(a**2 * th)
polar(th,r)
polar(th, -r)
show()
```

4.3.4 Polar Rose

A rose or rhodonea curve is a sinusoid $r = \cos(k\theta)$ plotted in polar coordinates. If k is an even integer, the curve will have $2k$ petals and k petals if it is odd. If k is rational, then the curve is closed and has finite length. If k is irrational, then it is not closed and has infinite length.

Example rose.py

```
from pylab import *
k = 4
th = linspace(0, 10*pi,1000)
r = cos(k*th)
polar(th,r)
show()
```

There are dozens of other famous curves whose details are available on the Internet. It may be an interesting exercise for the reader. For more details refer to [3, 2, 5]on the Internet.

4.4 Power Series

Trigonometric functions like sine and cosine sounds very familiar to all of us, due to our interaction with them since high school days. However most of us would find it difficult to obtain the numerical values of , say $\sin 5^0$, without trigonometric tables or a calculator. We know that differentiating a sine function twice will give you the original function, with a sign reversal, which implies
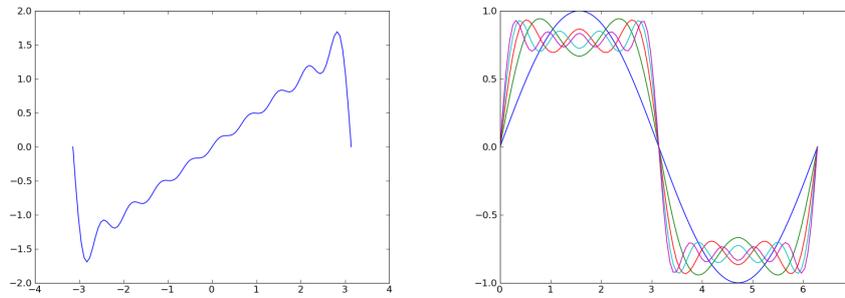
$$\frac{d^2y}{dx^2} + y = 0$$

Figure 4.7: Sawtooth and Square waveforms generated using Fourier series.

which has a series solution of the form

$$y = a_0 \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} + a_1 \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{4.5}$$

These are the Maclaurin series for sine and cosine functions. The following code plots several terms of the sine series and their sum.

Example series_sin.py

```
from pylab import *
from scipy import factorial
x = linspace(-pi, pi, 50)
y = zeros(50)
for n in range(5):
        term = (-1)**(n) * (x**(2*n+1)) / factorial(2*n+1)
        y = y + term
        #plot(x,term) #uncomment to see each term
plot(x, y, '+b')
plot(x, sin(x),'r') # compare with the real one
show()
```

The output of *series_sin.py* is shown in figure 4.6(a). For comparison the sin function from the library is plotted. The values calculated by using the series becomes closer to the actual value with more and more number of terms. The error can be obtained by adding the following lines to *series_sin.py* and the effect of number of terms on the error can be studied.

```
err = y - sin(x)
plot(x,err)
for k in err:
    print k
```

4.5   Fourier Series

A Fourier series is an expansion of a periodic function $f(x)$ in terms of an infinite sum of sines and cosines. The computation and study of Fourier series is known as harmonic analysis and is extremely useful as a way to break up an arbitrary periodic function into a set of simple terms that can be plugged in, solved individually, and then recombined to obtain the solution to the original problem or an approximation to it to whatever accuracy is desired or practical.

The examples below shows how to generate a square wave and sawtooth wave using this technique. To make the output better, increase the number of terms by changing the argument of the range() function, used in the for loop. The output of the programs are shown in figure 4.7.

Example fourier_square.py

```
from pylab import *
N = 100     # number of points
x = linspace(0.0, 2 * pi, N)
y = zeros(N)
for n in range(5):
    term = sin((2*n+1)*x) / (2*n+1)
```
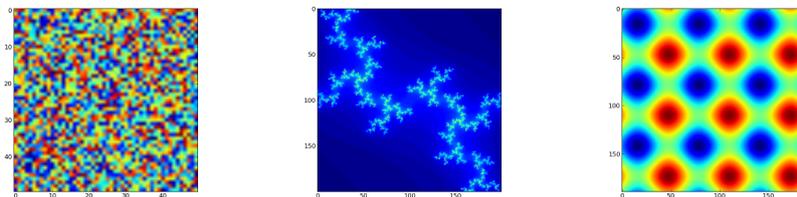
Figure 4.8: Outputs of (a) imshow1.py (b) julia.py (c) mgrid2.py

```
        y = y + term
        plot(x,y)
    show()
```

Example fourier_sawtooth.py

```
    from pylab import *
    N = 100      # number of points
    x = linspace(-pi, pi, N)
    y = zeros(N)
    for n in range(1,10):
        term = (-1)**(n+1) * sin(n*x) / n
        y = y + term
    plot(x,y)
    show()
```

4.6   2D plot using colors

A two dimensional matrix can be represented graphically by assigning a color to each point proportional to the value of that element. The program imshow1.py makes a 50×50 matrix filled with random numbers and uses *imshow()* to plot it. The result is shown in figure 4.8(a).

Example imshow1.py

```
    from pylab import *
    m = random([50,50])
    imshow(m)
    show()
```

4.7   Fractals

Fractals[3] are a part of fractal geometry, which is a branch of mathematics concerned with irregular patterns made of parts that are in some way similar to the whole (e.g.: twigs and tree branches). A fractal is a design of infinite details. It is created using a mathematical formula. No matter how closely you look at a fractal, it never loses its detail. It is infinitely detailed, yet it can be contained in a finite space. Fractals are generally self-similar and independent of scale. The theory of fractals was developed from Benoit Mandelbrot's study of complexity and chaos. Complex numbers are required to compute the Mandelbrot and Julia Set fractals and it is assumed that the reader is familiar with the basics of complex numbers.

To compute the basic Mandelbrot (or Julia) set one uses the equation $f(z) \rightarrow z^2 + c$, where both z and c are complex numbers. The function is evaluated in an iterative manner, ie. the result is assigned to z and the process is repeated. The purpose of the iteration is to determine the behavior of the values that are put into the function. If the value of the function goes to infinity (practically to some fixed value, like 1 or 2) after few iterations for a particular value of z , that point is considered to be outside the Set. A Julia set can be defined as the set of all the complex numbers (z) such that the iteration of $f(z) \rightarrow z^2 + c$ is bounded for a particular value of c.

To generate the fractal the number of iterations required to diverge is calculated for a set of points in the selected region in the complex plane. The number of iterations taken for diverging decides the color of each point. The points that did not diverge, belonging to the set, are plotted with the same color. The program *julia.py* generates a fractal using a julia set. The program creates a 2D array (200 x 200 elements). For our calculations, this array represents a rectangular region on the complex plane centered at the origin whose lower

---

[3]http://en.wikipedia.org/wiki/Fractal

left corner is (-1,-j) and the upper right corner is (1+j). For 200x200 equidistant points in this plane the number of iterations are calculated and that value is given to the corresponding element of the 2D matrix. The plotting is taken care by the imshow function. The output is shown in figure 4.8(b). Change the value of c and run the program to generate more patterns. The equation also may be changed.

Example julia.py

```
'''
Region of a complex plane ranging from -1 to +1 in both real
and imaginary axes is represented using a 2D  matrix
having X x Y elements.For X and Y equal to 200,the stepsize
in the complex plane is 2.0/200 = 0.01.
The nature of the pattern depends much on the value of c.
'''
from pylab import *
X = 200
Y = 200
MAXIT = 100
MAXABS = 2.0
c = 0.02 - 0.8j   # The constant in equation z**2 + c
m = zeros([X,Y])  # A two dimensional array

def numit(x,y):        # number of iterations to diverge
    z = complex(x,y)
    for k in range(MAXIT):
            if abs(z) <= MAXABS:
                z = z**2 + c
            else:
                return k       # diverged after k trials
    return MAXIT               # did not diverge,

for x in range(X):
    for y in range(Y):
        re = 0.01 * x - 1.0     # complex number for
        im = 0.01 * y - 1.0     # this (x,y) coordinate
        m[x][y] = numit(re,im)  # get the color for (x,y)
imshow(m)  # Colored plot using the 2D matrix
show()
```

4.8 Meshgrids

in order to make contour and 3D plots, we need to understand the meshgrid. Consider a rectangular area on the X-Y plane. Assume there are m divisions in the X direction and n divisions in the Y direction. We now have a $m \times n$ mesh. A meshgrid is the coordinates of a grid in a 2D plane, x coordinates of each mesh point is held in one matrix and y coordinates are held in another.

The NumPy function meshgrid() creates two 2x2 matrices from two 1D arrays, as shown in the example below. This can be used for plotting surfaces and contours, by assigning a Z coordinate to every mesh point.

Example mgrid1.py

```
from numpy import *
x = arange(0, 3, 1)
y = arange(0, 3, 1)
gx, gy = meshgrid(x, y)
print gx
print gy
```
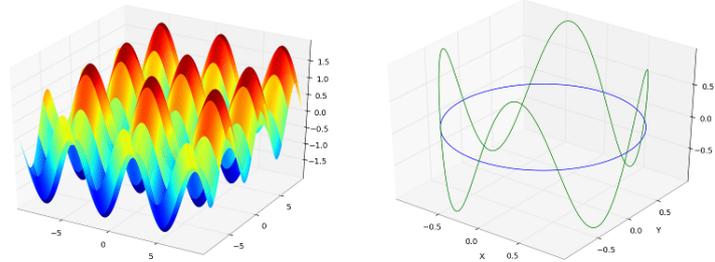
The outputs are as shown below, gx(i,j) contains the x-coordinate and gx(i,j) contains the y-coordinate of the point (i,j).

[[0 1 2]
[0 1 2]
[0 1 2]]
[[0 0 0]
[1 1 1]

Output of (a)surface3d.py (b)line3d.py

    [2 2 2]]
We can evaluate a function at all points of the meshgrid by passing the meshgrid as an argument. The program mgrid2.py plots the sum of sines of the x and y coordinates, using imshow to get a result as shown in figure 4.8(c).

Example mgrid2.py

```
from pylab import *
x = arange(-3*pi, 3*pi, 0.1)
y = arange(-3*pi, 3*pi, 0.1)
xx, yy = meshgrid(x, y)
z = sin(xx) + sin(yy)
imshow(z)
show()
```

4.9   3D Plots

Matplotlib supports several types of 3D plots, using the Axes3D class. The following three lines of code are required in every program making 3D plots using matplotlib.

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
ax = Axes3D(figure())
```

4.9.1   Surface Plots

The example mgrid2.py is re-written to make a surface plot using the same equation in surface3d.py and the result is shown in figure 4.9.1(a).

Example sufrace3d.py

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
ax = Axes3D(figure())
x = arange(-3*pi, 3*pi, 0.1)
y = arange(-3*pi, 3*pi, 0.1)
xx, yy = meshgrid(x, y)
z = sin(xx) + sin(yy)
ax.plot_surface(xx, yy, z, cmap=cm.jet, cstride=1)
show()
```

4.9.2   Line Plots

Example of a line plot is shown in line3d.py along with the output in figure 4.9.1(b).

Example line3d.py

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
ax = Axes3D(figure())
phi = linspace(0, 2*pi, 400)
x = cos(phi)
y = sin(phi)
```
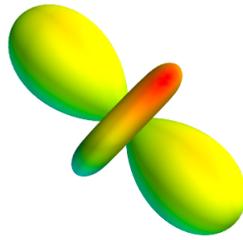
Figure 4.9: Output of ylm20.py

```
z = 0
ax.plot(x, y, z, label = 'x')# circle
z = sin(4*phi)  # modulated in z plane
ax.plot(x, y, z, label = 'x')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
show()
```

Modify the code to make x = sin(2*phi) to observe Lissajous figures

4.9.3   Wire-frame Plots

Data for a sphere is generated using the outer product of matrices and plotted, by sphere.py.

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
ax = Axes3D(figure())
phi = linspace(0, 2 * pi, 100)
theta = linspace(0, pi, 100)
x = 10 * outer(cos(phi), sin(theta))
y = 10 * outer(sin(phi), sin(theta))
z = 10 * outer(ones(size(phi)), cos(theta))
ax.plot_wireframe(x,y,z, rstride=2, cstride=2)
show()
```

4.10   Mayavi, 3D visualization

For more efficient and advanced 3D visualization, use Mayavi that is available on most of the GNU/Linux platforms. Program ylm20.py plots the spherical harmonics Y_m^l for $l = 2, m = 0$, using mayavi. The plot of $Y_2^0 = \frac{1}{4}\sqrt{\frac{5}{\pi}}(3\cos^2\phi - 1)$ is shown in figure4.9.

Example ylm20.py

```
from numpy import *
from enthought.mayavi import mlab
polar = linspace(0,pi,100)
azimuth = linspace(0, 2*pi,100)
phi,th = meshgrid(polar, azimuth)
r = 0.25 * sqrt(5.0/pi) * (3*cos(phi)**2 - 1)
x = r*sin(phi)*cos(th)
y = r*cos(phi)
z = r*sin(phi)*sin(th)
mlab.mesh(x, y, z)
mlab.show()
```

4.11 Exercises

1. Plot a sine wave using markers +, o and x using three different colors.

2. Plot $\tan\theta$ from θ from -2π to 2π, watch for singular points.

3. Plot a circle using the polar() function.

4. Plot the following from the list of Famous curves at reference [3]
   a) $r^2 = a^2 \cos 2\theta$, Lemniscate of Bernoulli
   b) $y = \sqrt{2\pi}e^{-x^2/2}$ Frequency curve
   c) $a\cosh(x/a)$ catenary
   d) $\sin(a\theta)$ for a = 2, 3, and 4. Rhodonea curves

5. Generate a triangular wave using Fourier series.

6. Evaluate $y = \sum_{n=1}^{n=\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$ for 10 terms.

7. Write a Python program to calculate sine function using series expansion and plot it.

8. Write a Python program to plot $y = 5x^2 + 3x + 2$ (for x from 0 to 5, 20 points),using pylab, with axes and title. Use red colored circles to mark the points.

9. Write a Python program to plot a Square wave using Fourier series, number of terms should be a variable.

10. Write a Python program to read the x and y coordinates from a file, in a two column format, and plot them.

11. Plot $x^2 + y^2 + z^2 = 25$ using mayavi.

12. Make a plot z = sin(x) + sin(y) using imshow() , from $-4\pi to 4\pi$ for both x and y.

13. Write Python code to plot $y = x^2$, with the axes labelled

Chapter 5

Type setting using LaTeX

LaTeX is a powerful typesetting system, used for producing scientific and mathematical documents of high typographic quality. LaTeX is not a word processor! Instead, LaTeX encourages authors not to worry too much about the appearance of their documents but to concentrate on getting the right content. You prepare your document using a plain text editor, and the formatting is specified by commands embedded in your document. The appearance of your document is decided by LaTeX, but you need to specify it using some commands. In this chapter, we will discuss some of these commands mainly to typeset mathematical equations. [1]

5.1 Document classes

LaTeX provides several predefined document classes (book, article, letter, report, etc.) with extensive sectioning and cross-referencing capabilities. Title, chapter, section, subsection, paragraph, subparagraph etc. are specified by commands and it is the job of LaTeX to format them properly. It does the numbering of sections automatically and can generate a table of contents if requested. Figures and tables are also numbered and placed without the user worrying about it.

The latex source document (the .tex file) is compiled by the latex program to generate a device independent (the .dvi file) output. From that you can generate postscript or PDF versions of the document. We will start with a simple example *hello.tex* to demonstrate this process. In a line, anything after a % sign is taken as a comment.

Example hello.tex

```
\documentclass{article}
\begin{document}
Small is beautiful.   % I am just a comment
\end{document}
```

Compile, view and make a PDF file using the following commands:
    $ latex hello.tex
    $ xdvi hello.dvi
    $ dvipdf hello.dvi

The output will look like : Small is beautiful.

5.2 Modifying Text

In the next example *texts.tex* we will demonstrate different types of text. We will \newline or \\ to generate a line break. A blank line will start a new paragraph.

Example texts.tex

```
\documentclass{article}
\begin{document}
This is normal text.
\newline
\textbf{This is bold face text.}
\textit{This is italic text.}\\
```

---

[1] http://www.latex-project.org/
http://mirror.ctan.org/info/lshort/english/lshort.pdf
http://en.wikibooks.org/wiki/LaTeX

Contents

1   Animals

This document defines sections.

1.1   Domestic

This document also defines subsections.

1.1.1   cats and dogs

Cats and dogs are domestic animals.

Figure 5.1: Output of sections.tex

\tiny{This is tiny text.}
\large{This is large text.}
\underline{This is underlined text.}
\end{document}

Compiling *texts.tex,* as explained in the previous example, will genearte the following output.

    This is normal text.
This is bold face text. *This is italic text.*
_{This is tiny text.} This is large text. <u>This is underlined text.</u>

5.3   Dividing the document

A document is generally organized in to sections, subsections, paragraphs etc. and Latex allows us to do this by inserting commands like section subsection etc. If the document class is book, you can have chapters also. There is a command to generate the table of contents from the sectioning information.[2]

Example sections.tex

> \documentclass{article}
> \begin{document}
> \tableofcontents
> \section{Animals}
> This document defines sections.
> \subsection{Domestic}
> This document also defines subsections.
> \subsubsection{cats and dogs}
> Cats and dogs are Domestic animals.
> \end{document}

The output of sections.tex is shown in figure 5.1.

---

[2]To generate the table of contents, you may have to compile the document two times.

5.4   Environments

Environments decide the way in which your text is formatted : numbered lists, tables, equations, quotations, justifications, figure, etc. are some of the environments. Environments are defined like :

\begin{environment_name} your text \end{environment_name}

The example program *environ.tex* demonstrates some of the environments.

Example environs.tex

\documentclass{article}
\begin{document}
\begin{flushleft} A bulleted list. \end{flushleft}
\begin{itemize} \item dog \item cat \end{itemize}
\begin{center} A numbered List. \end{center}
\begin{enumerate} \item dog \item cat \end{enumerate}
\begin{flushright} This text is right justified. \end{flushright}
\begin{quote}
Any text inside quote\\ environment will appe-\\ ar as typed.\\
\end{quote}
\begin{verbatim}
x = 1
while x <= 10:
   print x * 5
   x = x + 1
\end{verbatim}
\end{document}

The enumerate and itemize are used for making numbered and non-numbered lists. Flushleft, flushright and center are used for specifying text justification. Quote and verbatim are used for portions where we do not want LATEX to do the formatting. The output of environs.tex is shown below.

A bulleted list.

- dog

- cat

<div align="center">A numbered List.</div>

1. dog

2. cat

<div align="right">This text is right justified.</div>

   Any text inside quote
   environment will appe-
   ar as typed.

```
x = 1     # a Python program
while x <= 10:
   print x * 5
   x = x + 1
```

5.5   Typesetting Equations

There two ways to typeset mathematical formulae: in-line within a paragraph, or in a separate line. In-line equations are entered between *two $ symbols*. The equations in a separate line can be done within the *equation* environment. Both are demonstrated in math1.tex. *We use the amsmath package in this example.*

Example math1.tex

> \documentclass{article}
> \usepackage{amsmath}
> \begin{document}
> The equation $a^2 + b^2 = c^2$ is typeset as inline.
> The same can be done in a separate line using
> \begin{equation}
> a^2 + b^2 = c^2
> \end{equation}
> \end{document}

The output of this file is shown below.

The equation $a^2 + b^2 = c^2$ is typeset as inline. The same can be done in a separate line using

$$a^2 + b^2 = c^2 \tag{5.1}$$

The equation number becomes 5.1 because this happens to be the first equation in chapter 5.

5.5.1   Building blocks for typesetting equations

To typeset equations, we need to know the commands to make constructs like fraction, sqareroot, integral etc. The following list shows several commands and corresponding outputs. For each item, the output of the command, between the two $ signs, is shown on the right side. The reader is expected to insert then inside the body of a document, compile the file and view the output for practicing.

1. Extra space[3] : $A \quad B\qquad C$ $A \quad B \qquad C$

2. Greek letters : $ \alpha \beta \gamma \pi$ $\alpha\beta\gamma\pi$

3. Subscript and Exponents : $A_n \quad A^m $ $A_n \quad A^m$

4. Multiple Exponents : $a^b \quad a^{b^c}$ $a^b \quad a^{b^c}$

5. Fractions : $\frac{3}{5}$ $\frac{3}{5}$

6. Dots : $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ $n! = 1 \cdot 2 \cdots (n-1) \cdot n$

7. Under/over lines : $0.\overline{3} = \underline{1/3}}$ $0.\overline{3} = \underline{1/3}$

8. Vectors : $\vec{a}$ $\vec{a}$

9. Functions : $\sin x + \arctan y$ $\sin x + \arctan y$

10. Square root : $\sqrt{x^2+y^2}$ $\sqrt{x^2 + y^2}$

11. Higher roots : $z=\sqrt[3]{x^{2} + \sqrt{y}}$ $z = \sqrt[3]{x^2 + \sqrt{y}}$

12. Equalities : $A \neq B \quad A \approx C$ $A \neq B \quad A \approx C$

---

[3]\quad is for inserting space, the size of a \quad corresponds to the width of the character 'M' of the current font. Use \qquad for larger space.

13. Arrows : $\Leftrightarrow\quad\Downarrow$ $\Leftrightarrow \quad \Downarrow$

14. Partial derivative : $\frac{\partial ^2A}{\partial x^2}$ $\frac{\partial^2 A}{\partial x^2}$

15. Summation : $\sum_{i=1}^n$ $\sum_{i=1}^n$

16. Integration : $\int_0^{\frac{\pi}{2}} \sin x$ $\int_0^{\frac{\pi}{2}} sinx$

17. Product : $\prod_\epsilon$ $\prod_\epsilon$

18. Big brackets : $\Big((x+1)(x-1)\Big)^{2}$ $\Big((x+1)(x-1)\Big)^2$

19. Integral : $\int_a^b f(x) dx$ $\int_a^b f(x)dx$

20. Operators : $\pm \div \times \cup \ast$ $\pm \div \times \cup \ast$

5.6   Arrays and matrices

To typeset arrays, use the array environment, that is similar to the tabular environment. Within an array environment, & character separates columns, \\ starts a new line. The command \hline inserts a horizontal line. Alignment of the columns is shown inside braces using characters (lcr) and the | symbol is used for adding vertical lines. An example of making a table is shown below.

```
$ \begin{array}{|l|cr|}\hline
person & sex & age \\
John & male & 20 \\
Mary & female & 10 \\
Gopal & male & 30 \\
\hline
\end{array} $
```

person	sex	age
John	male	7
Mary	female	20
Gopal	male	30

The first column is left justified, second is centered and the third is right justified (decided by the {|l|cr|}). If you insert a | character between c and r, it will add a vertical line between second and third columns.

Let us make a matrix using the same command.

```
$ A = \left(
\begin{array}{ccc}
x_1 & x_2 & \ldots \\
y_1 & y_2 & \ldots \\
\vdots & \vdots & \ddots \\
\end{array}
\right) $
```

The output is shown below. The \left(and \right) provides the enclosure. All the columns are centered. We have also used horizontal, vertical and diagonal dots in this example.

$$A = \begin{pmatrix} x_1 & x_2 & \cdots \\ y_1 & y_2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

5.7   Floating bodies, Inserting Images

Figures and tables need special treatment, because they cannot be broken across pages. One method would be to start a new page every time a figure or a table is too large to fit on the present page. This approach would leave pages partially empty, which looks very bad. The easiest solution is to *float* them and let LATEX decide the position. (You can influence the placement of the floats using the arguments [htbp], here, top, bottom or special page). Any material enclosed in a figure or table environment will be treated as floating matter. The *graphicsx* packages is required in this case.

```
\usepackage{graphicx}
\text{Learn how to insert pictures with caption inside the figure environment.}
\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{pics/arcs.eps}
\includegraphics[width=0.2\textwidth]{pics/sawtooth.eps}
\caption{Picture of Arc and Sawtooth, inserted with [h] option.}
\end{figure}
```

The result is shown below.

Learn how to insert pictures with caption inside the figure environment.
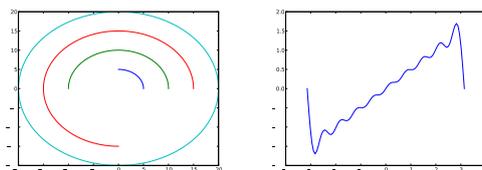

Figure 5.2: Picture of Arc and Sawtooth, inserted with [h] option.

5.8   Example Application

Latex source code for a simple question paper listed below.

Example qpaper.tex

```
\documentclass{article}
\usepackage{amsmath}
begin{document}
\begin{center}
\large{\textbf{Sample Question Paper\\for\\
Mathematics using Python}}
\end{center}
\begin{tabular}{p{8cm}r}
\textbf{Duration:3 Hrs} & \textbf{30 weightage}
\end{tabular}\\
\section{Answer all Questions. $4\times 1\frac{1}{2}$}
\begin{enumerate}
\item What are the main document classes in LATEX.
\item Typeset $\sin^{2}x+\cos^{2}x=1$ using LATEX.
\item Plot a circle using the polar() function.
\item Write code to print all perfect cubes upto 2000.
\end{enumerate}
```

```
\section{Answer any two Questions. $3\times 5$}
\begin{enumerate}
\item Set a sample question paper using LaTeX.
\item Using Python calculate the GCD of two numbers
\item Write a program with a Canvas and a circle.
\end{enumerate}
\begin{center}\text{End}\end{center}
\end{document}
```

The formatted output is shown below.

Sample Question Paper
for
Mathematics using Python

Duration:3 Hrs   **30 weightage**

1   Answer all Questions. $4 \times 1\frac{1}{2}$

1. What are the main document classes supported by LaTeX.

2. Typeset $\sin^2 x + \cos^2 x = 1$ using LaTeX.

3. Plot a circle using the polar() function.

4. Write code to print all perfect cubes upto 2000.

2   Answer any two Questions. 3×5

1. Set a sample question paper using LaTeX.

2. Write a Python function to calculate the GCD of two numbers

3. Write a program with a Canvas and a circle drawn on it.

End

5.9   Exercises

1. What are the main document classes supported by LaTeX.

2. How does Latex differ from other word processor programs.

3. Write a .tex file to typeset 'All types of Text Available' in tiny, large, underline and italic.

4. Rewrite the previous example to make the output a list of numbered lines.

5. Generate an article with section and subsections with table of contents.

6. Typeset 'All types of justifications' to print it three times; left, right and centered.

7. Write a .tex file that prints 12345 in five lines (one character per line).

8. Typeset a Python program to generate the multiplication table of 5, using verbatim.

9. Typeset $\sin^2 x + \cos^2 x = 1$

10. Typeset $\left(\sqrt{x^2 + y^2}\right)^2 = x^2 + y^2$

11. Typeset $\sum_{n=1}^{\infty} \left(1 + \frac{1}{n}\right)^n$

12. Typeset $\frac{\partial A}{\partial x} = A$

13. Typeset $\int_0^\pi \cos x.dx$

14. Typeset $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

15. Typeset $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

16. Typeset $R = \begin{pmatrix} \sin\theta & \cos\theta \\ \cos\theta & \sin\theta \end{pmatrix}$

Chapter 6

Numerical methods

Solving mathematical equations is an important requirement for various branches of science but many of them evade an analytic solution. The field of numerical analysis explores the techniques that give approximate but accurate solutions to such problems.[1] Even when they have a solution, for all practical purposes we need to evaluate the numeric value of the result, with the desired accuracy. We will focus on developing simple working programs rather than going into the theoretical details. The mathematical equations giving numerical solutions will be explored by changing various parameters and nature of input data.

6.1 Derivative of a function

The mathematical definition of the derivative of a function $f(x)$ at point x can be approximated by equation

$$\lim_{\triangle x \to 0} \frac{f(x + \frac{\triangle x}{2}) - f(x - \frac{\triangle x}{2})}{\triangle x} \tag{6.1}$$

neglecting the higher order terms. The accuracy of the derivative calculated using discrete values depends on the stepsize $\triangle x$. It will also depends on the number of higher order derivatives the function has. We will try to explore these aspects using the program *diff.py* , which evaluates the derivatives of few functions using two different stepsizes (0.1 ans 0.01). The input values to function deriv() are the function to be differentiated, the point at which the derivative is to be found and the stepsize $\triangle x$.

Example diff.py

```
def f1(x):
        return x**2
def f2(x):
        return x**4
def f3(x):
        return x**10

def deriv(func, x, dx=0.1):
        df = func(x+dx/2)-func(x-dx/2)
        return df/dx

print deriv(f1, 1.0), deriv(f1, 1.0, 0.01)
print deriv(f2, 1.0), deriv(f2, 1.0, 0.01)
print deriv(f3, 1.0), deriv(f3, 1.0, 0.01)
```

The output of the program is shown below. Comparing the two numbers on the same line shows the effect of stepsize. Comparing the first number on each row shows the effect of the

---

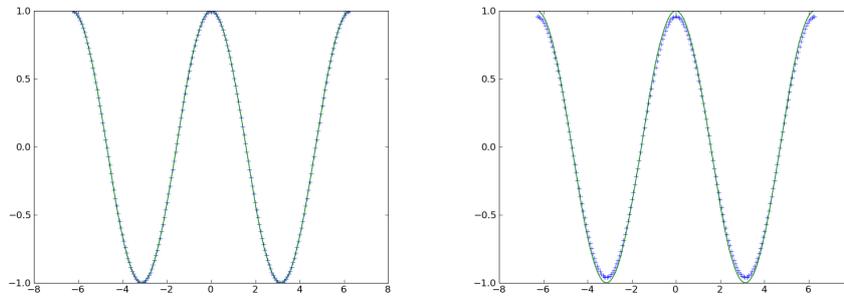[1] Introductory methods of numerical analysis by S.S.Sastry
http://ads.harvard.edu/books/1990fnmd.book/

Figure 6.1: Outputs of vdiff.py (a)for $\triangle x = 0.005$ (b) for $\triangle x = 1.0$

number of higher order derivatives the function has. For the same stepsize x^4 gives much less error than x^{10}. Second derivative of x^2 is constant and the result becomes exact, result on the first line.

        2.0 2.0
        4.01 4.0001
        10.3015768754 10.0030001575

You may explore other functions by modifying the program. It can be seen that the function deriv(), evaluates the function at two points to calculate the derivative. The higher order terms can be calculated by evaluating the function at more points. Techniques used for this will be discussed in section 6.8, on interpolation.

6.1.1   Differentiate Sine to get Cosine

The program *diff.py* in the previous example can only calculate the value of the derivative at a given point. In the program *vdiff.py*, we use a vectorized version of our deriv() function. The defined function is sine and the derivative is calculated using the vectorized version of deriv(). The actual cosine function also is plotted for comparison. The output of vdiff.py is shown in 6.1(a).

The value of $\triangle x$ is increased to 1.0 by changing one line of code as $y = vecderiv(x, 1.0)$ and the result is shown in 6.1(b). The values calculated using our function is shown using +marker, while the continuous curve is the expected result , ie. the cosine curve.

Example vdiff.py

```
from pylab import *
def f(x):
        return sin(x)

def deriv(x,dx=0.005):
        df = f(x+dx/2)-f(x-dx/2)
        return df/dx

vecderiv = vectorize(deriv)
x = linspace(-2*pi, 2*pi, 200)
y = vecderiv(x)
plot(x,y,'+')
plot(x,cos(x))
show()
```

6.2   Numerical Integration

Numerical integration constitutes a broad family of algorithms for calculating the numerical value of a definite integral. The objective is to find the area under the curve as shown in figure
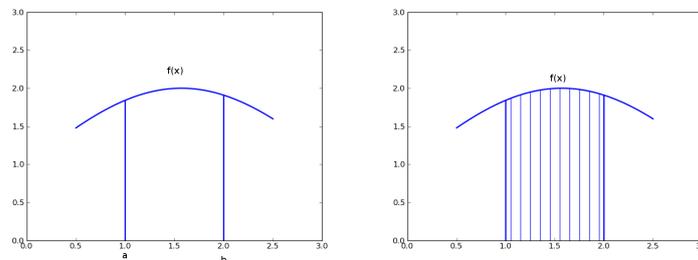
Figure 6.2: Area under the curve is divided it in to a large number of intervals. Area of each of them is calculated by assuming them to be trapezoids.

6.2. One method is to divide this area in to large number of sub-intervals and find the sum of their areas. The interval $a \leq x \leq b$ is divided in to n sub-intervals, each of length $h = (b-a)/n$, and area of a sub-interval is approximated by

$$\int_{x_{n-1}}^{x_n} y dx = \frac{h}{2}(y_{n-1} + y_n)$$

the integral is given by

$$\int_a^b y dx = \frac{h}{2}\left[y_0 + 2(y_1 + y_2 + \ldots + y_{n-1}) + y_n\right] \tag{6.2}$$

This is the sum of the areas of the individual trapezoids. The error in using the trapezoid rule is approximately proportional to $1/n^2$. If the number of sub-intervals is doubled, the error is reduced by a factor of 4. The program *trapez.py* does integration of a given function using equation 6.2. We will choose an example where the results can be cross checked easily, the value of π is calculated by evaluating the area of a unit circle by integrating the equation of a circle.

Example trapez.py

```
from math import *
def y(x): # equation of a circle
    return sqrt(1.0 - x**2)

def trapez(f, a, b, n):
    h = (b-a) / n
    sum = 0
    x = 0.5 * h    # f(x) at middle of the slice
    for i in range (1,n):
        sum = sum + h * f(x)
        x = x + h
    return sum

print 4 * trapez(y, 0.0, 1.0,1000)
print 4 * trapez(y, 0.0, 1.0,10000)
print trapez(sin,0,2,1000)    # Why the error ?
```

The output is shown below. The result gets better by increasing n thus resulting in smaller h. The last line shows, how things can go wrong if the arguments are given in the integer format. Learn how to avoid such pitfalls while writing programs. It is left as an exercise to the reader to modify the function trapez() to accept integer arguments also.
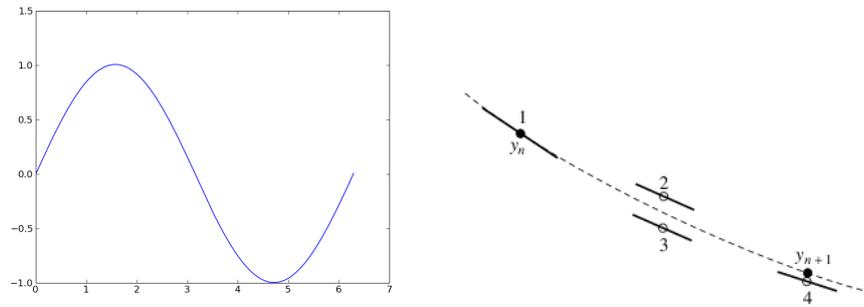
3.14041703178
3.14155546691
0.0

Figure 6.3: (a)Output of euler.py (b)Four intermediate steps of RK4 method

6.3 Ordinary Differential Equations

Differential equations are one of the most important mathematical tools used in producing models for physical and biological processes. In this section, we will discuss the numerical methods for solving the initial value problem for first-order ordinary differential equations. Consider the equation,

$$\frac{dy}{dx} = f(x, y); \quad y(x_0) = y_0 \tag{6.3}$$

where the derivative of the function $f(x, y)$ is known and the value of the function at some value of $x = x_0$ also is known. The objective is to find out the value of the function for other values of x. The underlying idea of any routine for solving the initial value problem is to rewrite the dy and dx as finite steps $\triangle y$ and $\triangle x$, and multiply the equations by $\triangle x$. This gives algebraic formulas for the change in the value of $y(x)$ when x is changed by one stepsize $\triangle x$. In the limit of making the stepsize very small, a good approximation to the underlying differential equation is achieved.

Implementation of this procedure results in the Euler's method, which is conceptually very important, but not recommended for any practical use. In this section we will discuss Euler's method and the Runge-Kutta method with the help of example programs. For detailed information refer to [10, 11].

6.3.1 Euler method

The equations of Euler's method can be obtained as follows. By the definition of derivative,

$$y'(x_n, y_n) = \frac{lim}{h \to 0} \frac{y(x_n + h) - y(x_n)}{h} \tag{6.4}$$

For sufficiently small values of h , we can write,

$$y(x_n + h) = y(x_n, y_n) + hy'(x_n) \tag{6.5}$$

The above equations implies that, if the value of the function $y(x)$ is known to be y_n at the point x_n, its value at a nearby point x_{n+1} is given by $y_n + h \times y'$. The program *euler.py* calculates the value of sine function using its derivative, ie. the cosine function. We start from $x = 0$, where $sin(x) = 0$ and compute the subsequent values using the derivative, $cos(x)$, and compare the result with the actual sine function.

Example euler.py

```
from pylab import *
h = 0.01    # stepsize
x = 0.0     # initial values
y = 0.0
ax = []   # Lists to store x and y
```
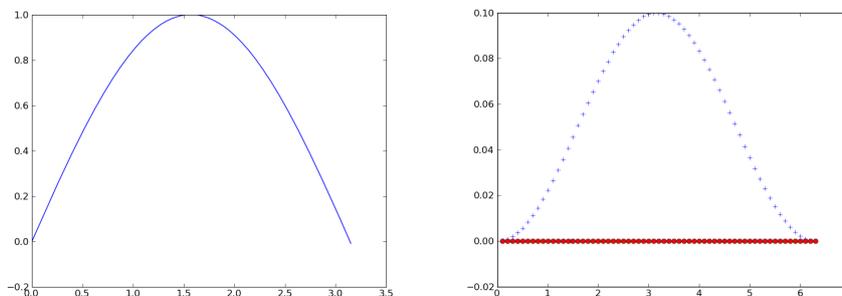
Figure 6.4: Outputs of (a)rk4.py (b) compareEuRK4.py

```
ay = []
while x < 2*pi:
    y = y + h * math.cos(x)    # Euler equation
    x = x + h
    ax.append(x)
    ay.append(y)
plot(ax,ay)
show()
```

The output of *euler.py* is shown in figure 6.3.

6.3.2 Runge-Kutta method

The formula 6.4 used by Euler method which advances a solution from x_n to x_{n+1} is not symmetric, it advances the solution through an interval h, but uses derivative information only at the beginning of that interval. Better results are obtained if we take *trial* step to the midpoint of the interval and use the value of both x and y at that midpoint to compute the *real* step across the whole interval. This is called the second-order Runge-Kutta or the midpoint method. This procedure can be further extended to higher orders.

The fourth order Runge-Kutta method is the most popular one and is commonly referred as the Runge-Kutta method. In each step the derivative is evaluated four times as shown in figure 6.4(a). Once at the initial point, twice at trial midpoints, and once at a trial endpoint. Every trial evaluation uses the value of the function from the previous trial point, ie. k_2 is evaluated using k_1 and not using y_n. From these derivatives the final function value is calculated, The calculation is done using the equations,

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2})$$
$$k_3 = hf(x_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2})$$
$$k_4 = hf(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{6.6}$$

The program rk4.py listed below uses the equations shown above to calculate the sine function, by integrating the cosine. The output is shown in figure 6.4(a).

Example rk4.py

```
from pylab import *

def rk4(x, y, fx, h = 0.1):    # x, y , f(x), stepsize
    k1 = h * fx(x)
    k2 = h * fx(x + h/2.0)
    k3 = h * fx(x + h/2.0)
```

```
        k4 = h * fx(x + h)
        return y + ( k1/6 + k2/3 + k3/3 + k4/6 )


    h = 0.01      # stepsize
    x = 0.0       # initial values
    y = 0.0
    ax = [x]
    ay = [y]
    while x < math.pi:
        y = rk4(x,y,math.cos)
        x = x + h
        ax.append(x)
        ay.append(y)
    plot(ax,ay)
    show()
```

The program compareEuRK4.py calculates the values of Sine by integrating Cosine. The errors
in both cases are evaluated at every step, by comparing it with the sine function, and plotted as
shown in figure 6.4(b). The accuracy of Runge-Kutta method is far superior to that of Euler's
method, for the same step size.

Example compareEuRK4.py

```
    from scipy import *

    def rk4(x, y, fx, h = 0.1):   # x, y , f(x), stepsize
        k1 = h * fx(x)
        k2 = h * fx(x + h/2.0)
        k3 = h * fx(x + h/2.0)
        k4 = h * fx(x + h)
        return y + ( k1/6 + k2/3 + k3/3 + k4/6 )


    h = 0.1     # stepsize
    x = 0.0     # initial values
    ye = 0.0    # for Euler
    yr = 0.0    # for RK4
    ax = []     # Lists to store results
    euerr = []
    rkerr = []
    while x < 2*pi:
        ye = ye + h * math.cos(x)   # Euler method
        yr = rk4(x, yr, cos, h)     # RK4 method
        x = x + h
        ax.append(x)
        euerr.append(ye - sin(x))
        rkerr.append(yr - sin(x))
    plot(ax,euerr,'o')
    plot(ax, rkerr,'+')
    show()
```

6.3.3 Function depending on the integral

In the previous section, the program *rk4.py* implemented a simplified version of the Runge-
Kutta method, the function was assumed to depend on the independent variable only. The
program *rk4_proper.py* listed below implements it properly. The functions $f(x,y) = 1 + y^2$
and $f(x,y) = (y - x)/y + x)$ are used for testing. Readers may verify the results by manual
computing.

Example rk4_proper.py

```
def f1(x,y):
    return 1 + y**2

def f2(x,y):
    return (y-x)/(y+x)

def rk4(x, y, fxy, h):  # x, y , f(x,y), step
    k1 = h * fxy(x, y)
    k2 = h * fxy(x + h/2.0, y+k1/2)
    k3 = h * fxy(x + h/2.0, y+k2/2)
    k4 = h * fxy(x + h, y+k3)
    ny = y + ( k1/6 + k2/3 + k3/3 + k4/6 )
    #print x,y,k1,k2,k3,k4, ny
    return ny

h = 0.2   # stepsize
x = 0.0   # initial values
y = 0.0
print rk4(x,y, f1, h)

h = 1
x = 0.0    # initial values
y = 1.0
print rk4(x,y,f2,h)
```

The results are shown below.

```
0.202707408081
1.5056022409
```

6.4   Polynomials

A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients. A polynomial in one variable with constant coefficients is given by

$$a_n x^n + ... + a_2 x^2 + a_1 x + a_0 \tag{6.7}$$

The derivative of 6.7 is,

$$n a_n x^{n-1} + ... + 2a_2 x + a_1$$

It is easy to find the derivative of a polynomial. Complicated functions can be analyzed by approximating them with polynomials. Taylor's theorem states that any sufficiently smooth function can locally be approximated by a polynomial. Computers use this property to evaluate trigonometric, logarithmic and exponential functions.

One dimensional polynomials can be explored using the *poly1d* function from Numpy. You can define a polynomial by supplying the coefficient as a list. For example , the statement p = poly1d([3,4,7]) constructs the polynomial $3x^2 + 4x + 7$. Numpy supports several polynomial operations. The following example demonstrates evaluation at a particular value, multiplication, differentiation, integration and division of polynomials using *poly1d*.

Example poly.py

```
from pylab import *
a = poly1d([3,4,5])
b = poly1d([6,7])
```
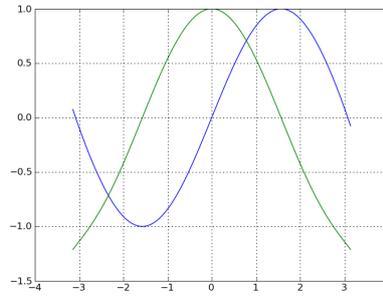
Figure 6.5: Output of polyplot.py

```
c = a * b + 5
d = c/a
print a
print a(0.5)
print b
print a * b
print a.deriv()
print a.integ()
print d[0], d[1]
```

The output of *poly.py* is shown below.

$3x^2 + 4x + 5$

7.75

$6x + 7$

$18x^3 + 45x^2 + 58x + 35$

$6x + 4$

$1x^3 + 2x^2 + 5x$

$6x + 7$

5

The last two lines show the result of the polynomial division, quotient and reminder. Note that a polynomial can take an array argument for evaluation to return the results in an array. The program *polyplot.py* evaluates polynomial 6.8 and its first derivative.

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} \tag{6.8}$$

The results are shown in figure 6.5. The equation 6.8 is the first four terms of series representing sine wave (7! = 5040). The derivative looks like cosine as expected. Try adding more terms and change the limits to see the effects.

Example polyplot.py

```
from pylab import *
x = linspace(-pi, pi, 100)
a = poly1d([-1.0/5040,0,1.0/120,0,-1.0/6,0,1,0])
da = a.deriv()
y = a(x)
y1 = da(x)
plot(x,y)
plot(x,y1)
show()
```

6.4.1   Taylor's Series

If a function and its derivatives are known at some point $x = a$, we can express $f(x)$ in the vicinity of that point using a polynomial. The Taylor series expansion is given by,

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \cdots + \frac{(x-a)^n}{n!}f^n(a) \qquad (6.9)$$

For example let us consider the equation

$$f(x) = x^3 + x^2 + x \qquad (6.10)$$

We can see that $f(0) = 0$ and the derivatives are

$$f'(x) = 3x^2 + 2x + 1; \qquad f''(x) = 6x + 2; \quad f'''(x) = 6$$

Using the values of the derivatives at $x = 0$ and equation 6.9, we evaluate the function at $x = .5$, using the polynomial expression,

$$f(0.5) = 0 + 0.5 \times 1 + \frac{0.5^2 \times 2}{2!} + \frac{0.5^3 \times 6}{3!} = .875$$

The result is same as $0.5^3 + 0.5^2 + 0.5 = .875$. We have calculated it manually for $x = .5$. We can also do this using Numpy as shown in the program taylor.py.

Example taylor.py

```
from numpy import *
p = poly1d([1,1,1,0])
dp = p.deriv()
dp2 = dp.deriv()
dp3 = dp2.deriv()
a = 0   # The known point
x = 0   # Evaluate at x
while x < .5:
    tay = p(a) + (x-a)* dp(a) + \
          (x-a)**2 * dp2(a) / 2 + (x-a)**3 * dp3(a)/6
    print '%5.1f  %8.5f\t%8.5f'%(x, p(x), tay)
    x = x + .1
```

The result is shown below.

```
0.0 0.00000 0.00000
0.1 0.11100 0.11100
0.2 0.24800 0.24800
0.3 0.41700 0.41700
0.4 0.62400 0.62400
```

6.4.2   Sine and Cosine Series

In the equation 6.9, let us choose $f(x) = sin(x)$ and $a = 0$. If $a = 0$, then the series is known as the Maclaurin Series. The first term will become $sin(0)$, which is just zero. The other terms involve the derivatives of $sin(x)$. The first, second and third derivatives of $sin(x)$ are $cos(x)$, $-sin(x)$ and $-cos(x)$, respectively. Evaluating each of these at zero, we get 1, 0 and -1 respectively. The terms with even powers vanish, resulting in,

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \qquad (6.11)$$

We can find the cosine series in a similar manner, to get

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} \qquad (6.12)$$

The program series_sc.py evaluates the sine and cosine series. The output is shown in figure 6.6. Compare the output of polyplot.py from section 6.4 with this. In both cases, we have evaluated the polynomial of sine function. In the present case, we can easily modify the number of terms and the logic is simpler.
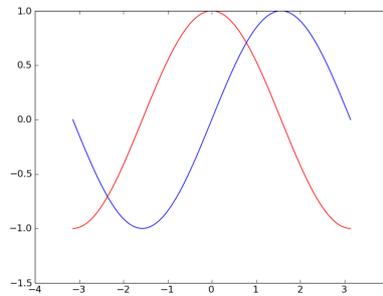
Figure 6.6: Output of series_sc.py

Example series_sc.py

```
from pylab import *

def f(n):      # Factorial function
    if n == 0:
        return 1
    else:
        return n * f(n-1)

NP = 100
x = linspace(-pi, pi, NP)
sinx = zeros(NP)
cosx = zeros(NP)
for n in range(10):
    sinx += (-1)**(n) * (x**(2*n+1)) / f(2*n+1)
    cosx += (-1)**(n) * (x**(2*n)) / f(2*n)
plot(x, sinx)
plot(x, cosx,'r')
show()
```

6.5   Finding roots of an equation

In general, an equation may have any number of roots, or no roots at all. For example $f(x) = x^2$ has a single root whereas $f(x) = sin(x)$ has an infinite number of roots. The roots can be located visually, by looking at the intersections with the x-axis. Another useful tool for detecting and bracketing roots is the incremental search method. The basic idea behind the incremental search method is simple: if $f(x1)$ and $f(x2)$ have opposite signs, then there is at least one root in the interval $(x1, x2)$. If the interval is small enough, it is likely to contain a single root. Thus the zeroes of $f(x)$ can be detected by evaluating the function at intervals of Δx and looking for change in sign.

There are several potential problems with the incremental search method: It is possible to miss two closely spaced roots if the search increment Δx is larger than the spacing of the roots. Certain singularities of $f(x)$ can be mistaken for roots. For example, $f(x) = tan(x)$ changes sign at odd multiples of $\pi/2$, but these locations are not true zeroes as shown in figure 6.7 (b).

Example *rootsearch.py* implements the function $root()$ that searches the roots of a function $f(x)$ from $x = a$ to $x = b$, incrementing it by dx.

Example rootsearch.py

```
def func(x):
    return x**3-10.0*x*x + 5
```
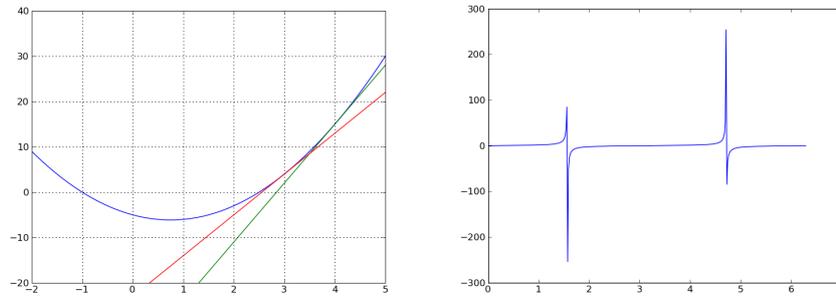
Figure 6.7: (a)Function $2x^2 - 3x - 5$ and its tangents at $x = 4$ and $x = 4$ (b) tan(x).

```
def root(f,a,b,dx):
    x = a
    while True:
        f1 = f(x)
        f2 = f(x+dx)
        if f1*f2 < 0:
            return x, x + dx
        x = x + dx
        if x >=  b:
            return (None,None)

x,y = root(func, 0.0, 1.0,.1)
print x,y
x,y = root(math.cos, 0.0, 4,.1)
print x,y
```

The outputs are $(0.7 , 0.8)$ and $(1.5 , 1.6)$. The root of cosine,$\pi/2$, is between 1.5 and 1.6. After the root has been located roughly, we can find the root with any specified accuracy, using bisection method, Newton-Raphson method etc.

6.5.1   Method of Bisection

The method of bisection finds the root by successively halving the interval until it becomes sufficiently small. Bisection is not the fastest method available for computing roots, but it is the most reliable. Once a root has been bracketed, bisection will always find it. The method of bisection works in the following manner. If there is a root between $x1$ and $x2$, then $f(x1) \times f(x2) < 0$. Next, we compute $f(x3)$, where $x3 = (x1 + x2)/2$. If $f(x2) \times f(x3) < 0$, then the root must be in $(x2, x3)$ and we replace the original bound $x1$ by $x3$. Otherwise, the root lies between $x1$ and $x3$, in that case $x2$ is replaced by $x3$. This process is repeated until the interval has been reduced to the specified value, say ε.

The number of bisections required to reach a prescribed limit, ε, is given by equation 6.13.

$$n = \frac{\ln(|\triangle x|)/\varepsilon}{\ln 2} \tag{6.13}$$

The program *bisection.py* finds the root of the equation $x^3 - 10x^2 + 5$. The starting values are found using the program *rootsearch.py*. The results are printed for two different accuracies.

Example bisection.py

```
import math def func(x):
        return x**3 - 10.0* x*x + 5

def bisect(f, x1, x2, epsilon=1.0e-9):
    f1 = f(x1)
```

```
        f2 = f(x2)
        if f1*f2 > 0.0:
            print 'x1 and x2 are on the same side of x-axis'
            return
        n = math.ceil(math.log(abs(x2 - x1)/epsilon)/math.log(2.0))
        n = int(n)
        for i in range(n):
            x3 = 0.5 * (x1 + x2)
            f3 = f(x3)
            if f3 == 0.0: return x3
            if f2*f3 < 0.0:
                x1 = x3
                f1 = f3
            else:
                x2 = x3
                f2 = f3
        return (x1 + x2)/2.0

    print bisect(func, 0.70, 0.8, 1.0e-4)
    print bisect(func, 0.70, 0.8, 1.0e-9)
```

6.5.2 Newton-Raphson Method

The Newton–Raphson algorithm requires the derivative of the function also to evaluate the roots. Therefore, it is usable only in problems where $f'(x)$ can be readily computed. It does not require the value at two points to start with. We start with an initial guess which is reasonably close to the true root. Then the function is approximated by its tangent line and the x-intercept of the tangent line is calculated. This value is taken as the next guess and the process is repeated. The Newton-Raphson formula is shown below.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{6.14}$$

Figure 6.7(a) shows the graph of the quadratic equation $2x^2 - 3x - 5 = 0$ and its two tangents. It can be seen that the zeros are at x = -1 and x = 2.5, and we use the program newraph.py shown below to find the roots. The function nr() is called twice, and we get the roots nearer to the corresponding starting values.

Example newraph.py

```
    from pylab import *
    def f(x):
        return 2.0 * x**2 - 3*x - 5

    def df(x):
        return 4.0 * x - 3

    def nr(x, tol = 1.0e-9):
        for i in range(30):
            dx = -f(x)/df(x)
            #print x
            x = x + dx
            if abs(dx) < tol:
                return x

    print nr(4)
    print nr(0)
```

The output is shown below.

2.5

-1.0

Uncomment the print statement inside nr() to view how fast this method converges, compared to the bisection method. The program newraph_plot.py, listed below is used for generating the figure 6.7.

Example newraph_plot.py

```python
from pylab import *
def f(x):
        return 2.0 * x**2 - 3*x - 5

def df(x):
        return 4.0 * x - 3

vf = vectorize(f)
x = linspace(-2, 5, 100)
y = vf(x)
# Tangents at x=3 and 4, using one point slope formula
x1 = 4
tg1 = df(x1)*(x-x1) + f(x1)
x1 = 3
tg2 = df(x1)*(x-x1) + f(x1)
grid(True)
plot(x,y)
plot(x,tg1)
plot(x,tg2)
ylim([-20,40])
show()
```

We have defined the function $f(x) = 2x^2 - 3x - 5$ and vectorized it. The derivative $4x^2 - 3$ also is defined by $df(x)$, which is the slope of $f(x)$. The tangents are drawn at $x = 4$ and $x = 3$, using the point slope formula for a line $y = m(x - x1) + y1$.

6.6   System of Linear Equations

A system of m linear equations with n unknowns can be written in a matrix form and can be solved by using several standard techniques like Gaussian elimination. In this section, the matrix inversion method, using Numpy, is demonstrated. For more information see reference [12].

6.6.1   Equation solving using matrix inversion

Non-homogeneous matrix equations of the form $Ax = b$ can be solved by matrix inversion to obtain $x = A^{-1}b$. The system of equations

$$4x + y - 2z = 0$$
$$2x - 3y + 3z = 9$$
$$-6x - 2y + z = 0$$

can be represented in the matrix form as

$$\begin{pmatrix} 4 & 1 & -2 \\ 2 & -3 & 3 \\ -6 & -2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 9 \\ 0 \end{pmatrix}$$

and can be solved by finding the inverse of the coefficient matrix.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 4 & 1 & -2 \\ 2 & -3 & 3 \\ -6 & -2 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 9 \\ 0 \end{pmatrix}$$

Using numpy we can solve this as shown in solve_eqn.py

Example solve_eqn.py

```
from numpy import *
b = array([0,9,0])
A = array([ [4,1,-2], [2,-3,3],[-6,-2,1]])
print dot(linalg.inv(A),b)
```

The result will be [0.75 -2. 0.5], that means $x = 0.75, y = -2, z = 0.5$. This can be verified by substituting them back in to the equations.

Exercise: solve x+y+3z = 6; 2x+3y-4z=6;3x+2y+7z=0

6.7   Least Squares Fitting

A mathematical procedure for finding the best-fitting curve $f(x)$ for a given set of points (x_n, y_n) by minimizing the sum of the squares of the vertical offsets of the points from the curve is called least squares fitting. The least square fit is obtained by minimizing the function,

$$S(a_0, a_1, \ldots, a_m) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 \tag{6.15}$$

with respect to each a_i and the condition for that is

$$\frac{\partial S}{\partial a_i} = 0, \quad i = 0, 1, \ldots m \tag{6.16}$$

For a linear fit, the equation is

$$f(a, b) = a + bx$$

Solving the equations $\frac{\partial S}{\partial a} = 0$ and $\frac{\partial S}{\partial b} = 0$ will give the result,

$$b = \frac{\sum y_i(x - \bar{x})}{\sum x_i(x - \bar{x})}, \quad and \quad a = \bar{y} - \bar{x}b \tag{6.17}$$

where $\bar{x}$ and $\bar{y}$ are the mean values defined by the equations,

$$\bar{x} = \frac{1}{n+1}\sum_{i=0}^{n} x_i, \quad \bar{y} = \frac{1}{n+1}\sum_{i=0}^{n} y_i \tag{6.18}$$

The program lsfit.py demonstrates the usage of equations 6.17 and 6.18.

Example lsfit.py

```
from pylab import *
NP = 50
r = 2*ranf([NP]) - 0.5
x = linspace(0,10,NP)
data = 3 * x + 2 + r

xbar = mean(x)
ybar = mean(data)
b = sum(data*(x-xbar)) / sum(x*(x-xbar))
```
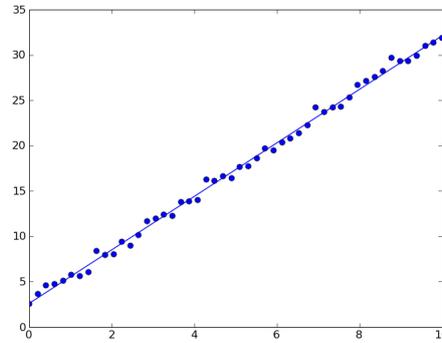
Figure 6.8: Output of lsfit.py

```
a = ybar - xbar * b
print a,b
y = a + b * x
plot(x,y)
plot(x,data,'ob')
show()
```

The raw data is made by adding random numbers (between -1 and 1) to the y coordinates generated by $y = 3*x + 2$. The Numpy functions mean() and sum() are used. The output is shown in figure 6.8.

6.8 Interpolation

Interpolation is the process of constructing a function $f(x)$ from a set of data points (x_i, y_i), in the interval $a < x < b$ that will satisfy $y_i = f(x_i)$ for any point in the same interval. The easiest way is to construct a polynomial of degree n that passes through the $n+1$ distinct data points.

6.8.1 Newton's polynomial

Suppose the the given set is $(x_i, y_i), i = 0, 1 \ldots n - 1$ and the polynomial is $P_n(x)$. Since the polynomial passes through all the data points, the following condition will be satisfied.

$$P_n(x_i) = y_i, \quad i = 0, 1 \ldots n - 1 \tag{6.19}$$

The Newton's interpolating polynomial is given by the equation,

$$P_n(x) = a_0 + (x - x_0)a_1 + \cdots + (x - x_0)\cdots(x - x_{n-1})a_n \tag{6.20}$$

The coefficients a_i can be evaluated in the following manner. When $x = x_0$, all the terms in 6.20 except a_0 will vanish due to the presence of $(x - x_0)$ and we get

$$y_o = a_0 \tag{6.21}$$

For $x = x_1$, only the first two terms will be non-zero.

$$y_1 = a_0 + a_1(x_1 - x_0) \tag{6.22}$$

$$a_1 = \frac{(y_1 - y_0)}{(x_1 - x_0)} \tag{6.23}$$

Applying $x = x_2$, we get

$$y_2 = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)a_1(x_2 - x_1) \tag{6.24}$$

$$a_2 = \frac{\frac{y_2-y_1}{x_2-x_1} - \frac{y_1-y_0}{x_1-x_0}}{x_2 - x_0} \qquad (6.25)$$

The other coefficients can be found in a similar manner. They can be expressed better using the divided difference notation as shown below.

$$[y_0] = y_0$$

$$[y_0, y_1] = \frac{(y_1 - y_0)}{(x_1 - x_0)}$$

$$[y_0, y_1, y_3] = \frac{\frac{y_2-y_1}{x_2-x_1} - \frac{y_1-y_0}{x_1-x_0}}{x_2 - x_0} = \frac{[y_1, y_2] - [y_0, y_1]}{(x_2 - x_0)}$$

Using these notation, the Newton's polynomial can be re-written as;

$$\begin{aligned} P(x) &= [y_0] + [y_0, y_1](x - x_0) + [y_0, y_1, y_2](x - x_0)(x - x_1) + \\ &\quad \cdots + [y_0, \ldots, y_n](x - x_0) \ldots (x - x_{n-1}) \end{aligned} \qquad (6.26)$$

The divided difference can be put in the tabular form as shown below. This will be useful while calculating the coefficients manually.

x_0	x_0	$[y_0]$			
x_1	y_1	$[y_1]$	$[y_0, y_1]$		
x_2	y_2	$[y_2]$	$[y_1, y_2]$	$[y_0, y_1, y_2]$	
x_3	y_3	$[y_3]$	$[y_2, y_3]$	$[y_1, y_2, y_3]$	$[y_0, y_1, y_2, y_3]$

0	0				
1	3	$\frac{3-0}{1-0} = 3$			
2	14	$\frac{14-3}{2-1} = 11$	$\frac{11-3}{2-0} = 4$		
3	39	$\frac{39-14}{3-2} = 25$	$\frac{25-11}{3-1} = 7$	$\frac{7-4}{3-0} = 1$	

The table given above shows the divided difference table for the data set x = [0,1,2,3] and y = [0,3,14,39], calculated manually. The program newpoly.py can be used for calculating the coefficients, which prints the output [0, 3, 4, 1].

Example newpoly.py

```
from copy import copy
def coef(x,y):
    a = copy(y)
    m = len(x)
    for k in range(1,m):
        tmp = copy(a)
        for i in range(k,m):
            tmp[i] = (a[i] - a[i-1])/(x[i]-x[i-k])
        a = copy(tmp)
    return a

x  = [0,1,2,3]
y  = [0,3,14,39]
print coef(x,y)
```

We start by copying the list y to coefficient a, the first element $a_0 = y_0$. While calculating the differences, we have used two loops and a temporary list. The same can be done in a better way using arrays of Numpy[2], as shown in newpoly2.py.

---
[2]This function is from reference [12], some PDF versions of this book are available on the web.
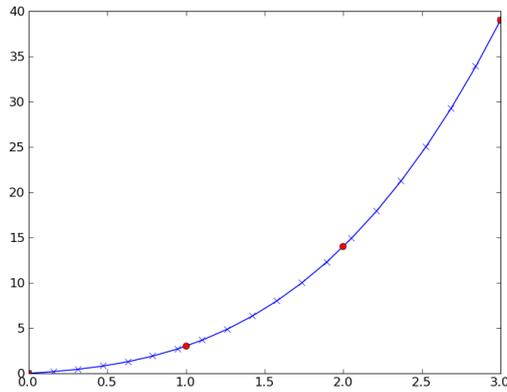
Figure 6.9: Output of newton_in3.py

Example newpoly2.py

```
from numpy import *
def coef(x,y):
    a = copy(y)
    m = len(x)
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(x[k:m]-x[k-1])
    return a

x  = array([0,1,2,3])
y  = array([0,3,14,39])
print coef(x,y)
```

The next step is to calculate the value of y for any given value of x, using the coefficients already calculated. The program *newinterpol.py* calculates the coefficients using the four data points. The function eval() uses the recurrence relation

$$P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, \ldots n \qquad (6.27)$$

The program generates 20 new values of x, and calculate corresponding values of y and plots them along with the original data points, as shown in figure 6.9.

Example newinterpol.py

```
from pylab import *
def eval(a,xpoints,x):
    n = len(xpoints) - 1
    p = a[n]
    for k in range(1,n+1):
        p = a[n-k] + (x -xpoints[n-k]) * p
    return p

def coef(x,y):
    a = copy(y)
    m = len(x)
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(x[k:m]-x[k-1])
    return a
```

```
x   = array([0,1,2,3])
y   = array([0,3,14,39])
coef = coef(x,y)
NP = 20
newx = linspace(0,3, NP)   # New x-values
newy = zeros(NP)
for i in range(NP): # evaluate y-values
    newy[i] = eval(coef, x, newx[i])
plot(newx, newy,'-x')
plot(x, y,'ro')
show()
```

You may explore the results for new points outside the range by changing the second argument of line *newx = linspace(0,3,NP)* to a higher value.

Look for similarities between Taylor's series discussed in section 6.4.1 that and polynomial interpolation process. The derivative of a function represents an infinitesimal change in the function with respect to one of its variables. The finite difference is the discrete analog of the derivative. Using the divided difference method, we are in fact calculating the derivatives in the discrete form.

6.9   Exercises

1. Differentiate $5x^2 + 3x + 5$ numerically and evaluate at $x = 2$ and $x = -2$.

2. Write code to numerically differentiate $\sin(x^2)$ and plot it by vectorizing the function. Compare with the analytical result.

3. Integrate $\ln x$, e^x from $x = 1$to 2.

4. Solve $2x + y = 3; -x + 4y = 0; 3 + 3y = -1$ using matrices.

5. Modify the program julia.py, $c = 0.2 - 0.8j$ and $z = z^6 + c$

6. Write Python code, using pylab, to solve the following equations using matrices
$$4x + y - 2z = 0$$
$$2x - 3y + 3z = 9$$
$$-6x - 2y + z = 0$$

7. Find the roots of $5x^2 + 3x - 6$ using bisection method.

8. Find the all the roots of $sin(x)$ between 0 and 10, using Newton-Raphson method.

Appendix A : Installing GNU/Linux

Programming can be learned better by practicing and it requires an operating system, and Python interpreter along with some library modules. All these requirements are packaged on the Live CD comes along with this book. You can boot any PC from this CD and start working. However, it is better to install the whole thing to a harddisk. The following section explains howto install GNU/Linux. We have selected the Ubuntu distribution due to its relatively simple installation procedure, ease of maintenanace and support for most of the hardware available in the market.

6.10   Installing Ubuntu

Most of the users prefer a dual boot system, to keep their MSWindows working. We will explain the installation process keeping that handicap in mind. All we need is an empty partition of minimum 5 GB size to install Ubuntu. Free space inside a Windows partition will not do, we need to format the partition to install Ubuntu. The Ubuntu installer will make the system multi-boot by searching through all the partitions for installed operating systems.

The System

This section will describe how Ubuntu was installed on a system, with MSWindows, having the following partitions:
C: (GNU/Linux calls it /dev/sda1) 20 GB
D: (/dev/sda5) 20 GB
E: (/dev/sda6) 30 GB
We will use the partition E: to install Ubuntu, it will be formatted.

The Procedure

Set the first boot device CD ROM from the BIOS. Boot the PC from the Ubuntu installation CD, we have used Phoenix Live CD (a modified version on Ubuntu 9.1). After 2 to 3 minutes a desktop as shown below will appear. Click on the Installer icon, the window shown next will pop up. Screens will appear to select the language, time zone and keyboard layout as shown in the figures below.
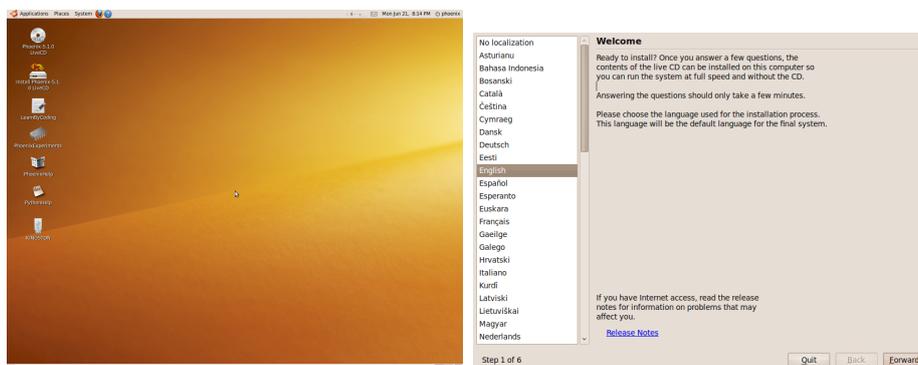
Now we proceed to the important part, choosing a partition to install Ubuntu.
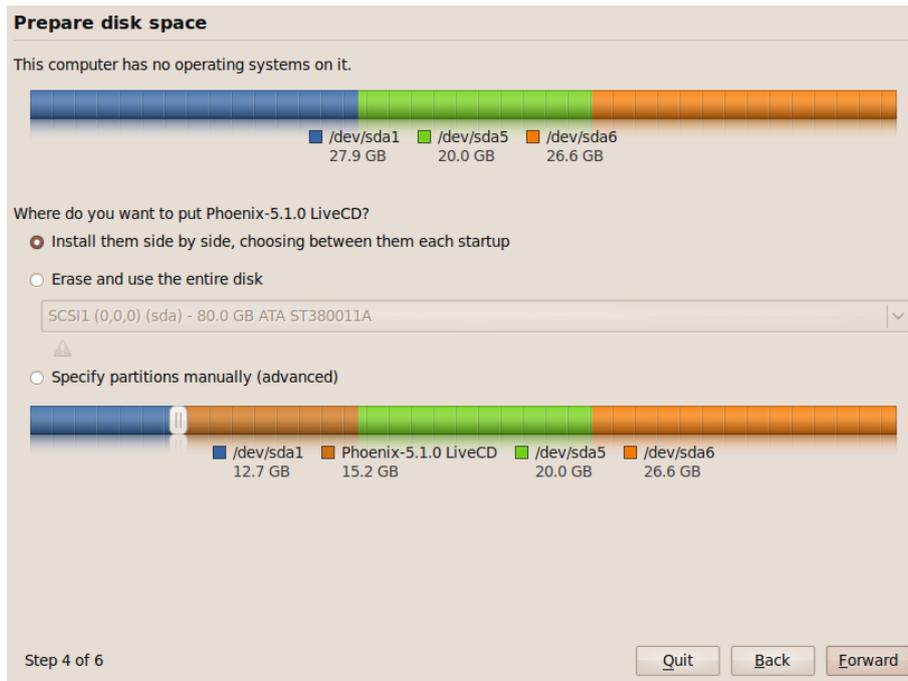

The bar on the top graphically displays the existing partitions. Below that there are three options provided :

1. Install them side by side.

2. Erase and use the entire disk.

3. Specify partitions manually.

If you choose the first option, the Installer will resize and repartition the disk to make some space for the new system. By default this option is marked as selected. The bar at the bottom shows the proposed partition scheme. In the present example, the installer plans to divide the C: drive in to two partitions to put ubuntu on the second.

We are going to choose the third option, choose the partition manually. We will use the last partition (drive E:) for installing Ubuntu. Once you choose that and click forward, a screen will appear where we can add, delete and change partitions. We have selected the third partition and clicked on Change. A pop-up window appeared. Using that we selected the file-system type to ext3, marked the format option, and selected the mount point as / . The screen with the pop-up window is shown below.
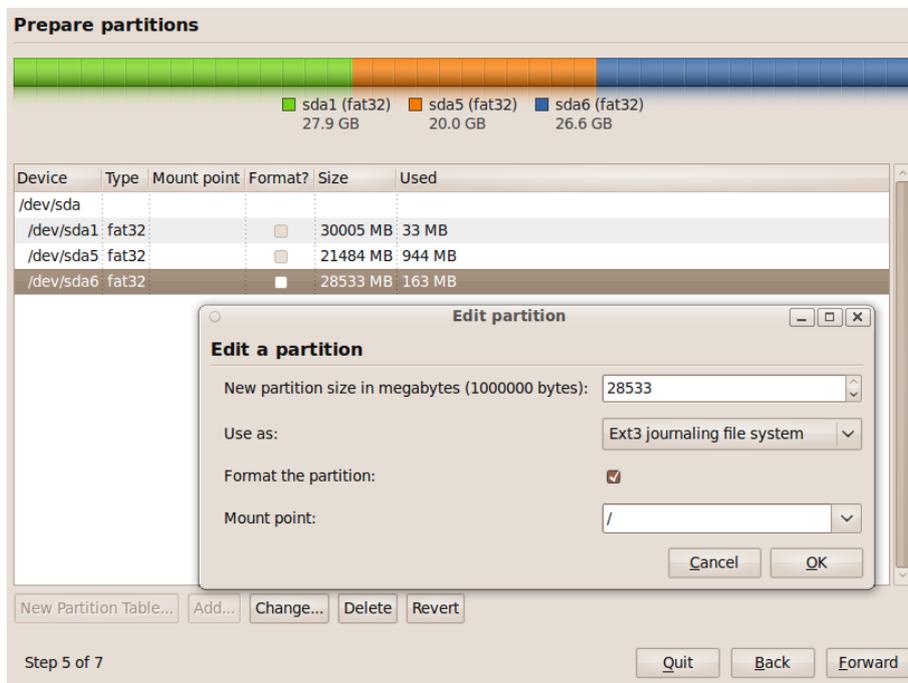
If we proceed with this, a warning will appear complaining about the absence of swap partitions. The swap partition is used for supplementing the RAM with some virtual memory. When RAM is full, processes started but not running will be swapped out. One can install a system without swap partition but it is a good idea to have one.

We decide to go back on the warning, to delete the E: drive, create two new partitions in that space and make one of them as swap. This also demonstrates how to make new partitions. The screen after deleting E: , with the pop-up window to make the swap partition is shown below.
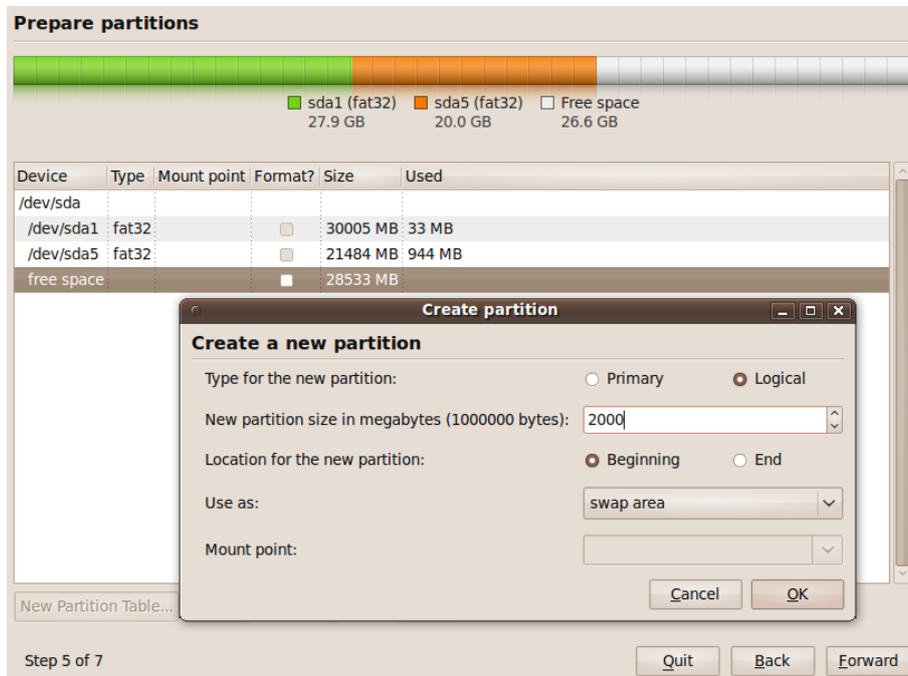
We made a 2 GB swap. The remaining space is used for making one more partition, as shown in the figure 6.10.

Once disk partitioning is over, you will be presented with a screen to enter a user name and password.[3] A warning will be issued if the password is less than 8 characters in length. You will be given an option to import desktop settings from other installations already on the disk,

---

[3]All GNU/Linux installations ask for a root password during installation. For simplicity, Ubuntu has decided to hide this information. The first user created during installation has special privileges and that password is asked, instead of the root password, for all system administration jobs, like installing new software.
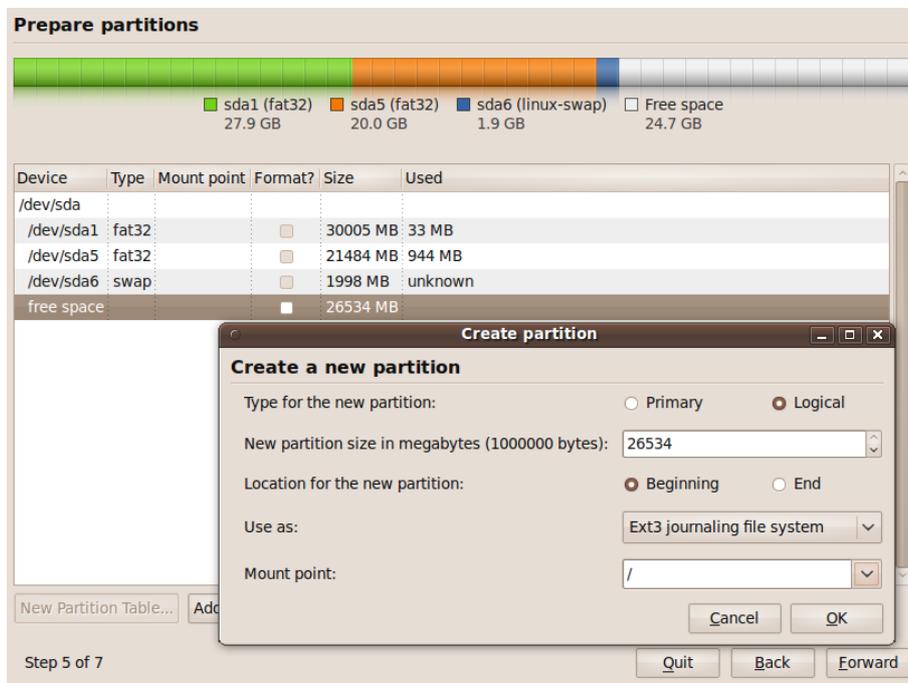
Figure 6.10: Making the partition to install Ubuntu.

choose this if you like. The next screen will confirm the installation. After the installation is over, mat take 10 to 15 minutes, you will be prompted to reboot the system. On rebooting you will be presented with a menu, to choose the operating system to boot. First item in the menu will be the newly installed Ubuntu.

6.11 Package Management

The Ubuntu install CD contains some common application programs like web browser, office package, document viewer, image manipulation program etc. After installing Ubuntu, you may want to add more applications. The Ubuntu repository has an enormous number of packages, that can be installed very easily. You need to have a reasonably fast Internet connection for this purpose.

From the main menu, open System->Administration->Synaptic package manager. After providing the pass word (of the first user, created during installation), the synaptic window will popup as shown in figure 6.11.

Select Settings->Repositories to get a pop-up window as shown below. Tick the four repositories, close the pop-up window and Click on Reload. Synaptic will now try to download the index files from all these repositories. It may take several minute.
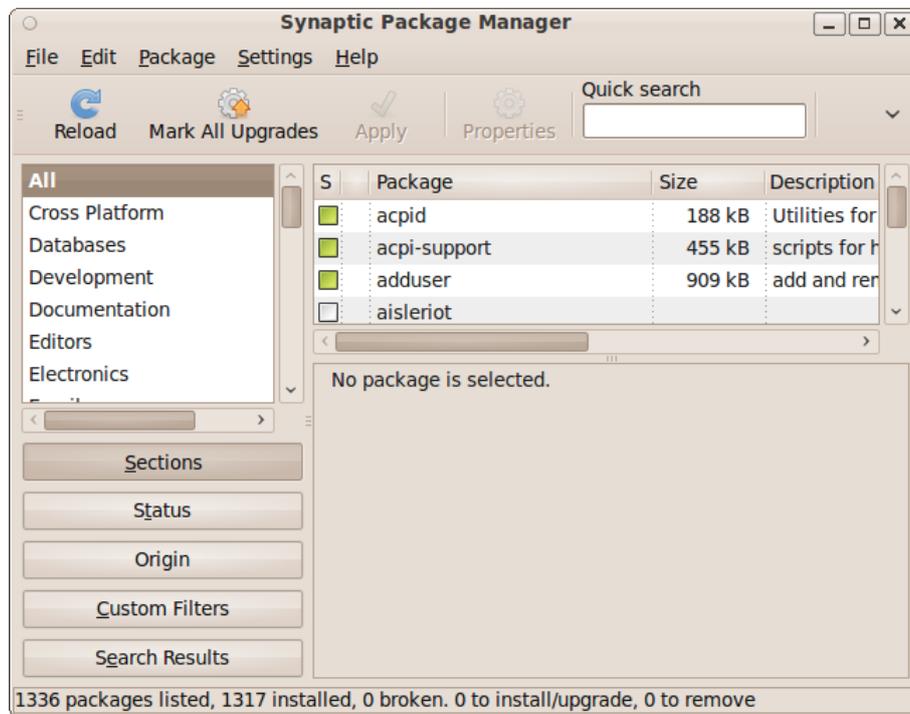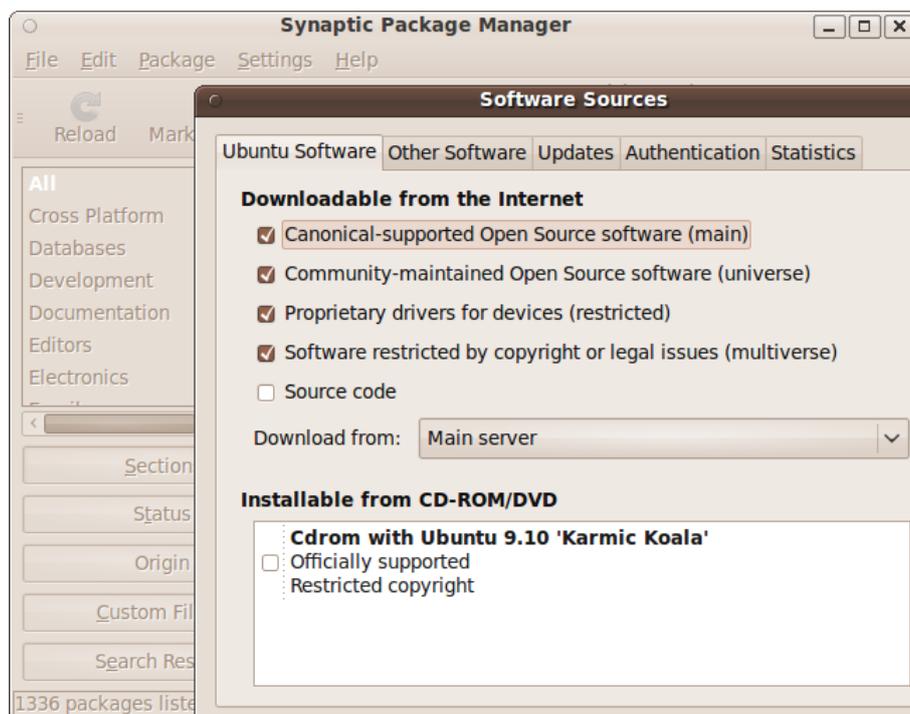
Figure 6.11: Synaptic package manager window

Now, you are ready to install any package from the Ubuntu repository. Search for any package by name, from these repositories, and install it. If you have done the installation from original Ubuntu CD, you may require the following packges:

- lyx : A latex front-end. Latex will be installed since lyx dpends on latex.

- python-matplotlib : The graphics library

- python-visual : 3D graphics

- python-imaging-tk : Tkinter, Python Imaging Library etc. will be installed

- build-essential : C compiler and related tools.

6.11.1 Install from repository CD

We can also install packages from repository CDs. Insert the CD in the drive and Select Add CDROM from the Edit menu of Synaptic. Now all the packages on the CD will be available for search and install.

6.11.1.1 Installing from the Terminal

You can install packages from the terminal also. You have to become the root user by giving the sudo command;

    $ sudo -s
    enter password :
    #

Note that the prompt changes from $ to #, when you become root.
Install packages using the command :

    #apt-cdrom add
    #apt-get install mayavi2

6.11.2 Behind the scene

Even though there are installation programs that performs all these steps automatically,it is better to know what is really happening. Installing an operating system involves;

- Partitioning of the hard disk

- Formatting the partitions

- Copying the operating system files

- Installing a boot loader program

The storage space of a hard disk drive can be divided into separate data areas, known as partitions. You can create primary partitions and extended partitions. Logical drives (secondary partitions can be created inside the extended partitions). On a disk, you can have up to 4 partitions, where one of them could be an extended partition. You can have many logical drives inside the extended partition.

On a MSWindows system, the primary partition is called the C: drive. The logical drives inside the extended partition are named from D: onwards. GNU/Linux uses a different naming convention. The individual disks are named as /dev/sda , /dev/sdb etc. and the partitions inside them are named as /dev/sda1, /dev/sda2 etc. The numbering of secondary partitions inside the logical drive starts at /dev/sda5. (1 to 4 are reserved for primary and extended). Hard disk partitioning can be done using the fdisk program. The installation program also does this for you.

The process of making a file system on a partition is called formatting. There are many different types of file systems. MSWindows use file systems like FAT32, NTFS etc. and GNU/Linux mostly uses file systems like ext3, ext4 etc.

The operating system files are kept in directories named boot, sbin, bin, etc etc. The kernel that loads while booting the system is kept in /boot. The configuration files are kept in /etc. /sbin and /bin holds programs that implements many of the shell commands. Most of the application programs are kept in /usr/bin area.

The boot loader program is the one provides the selection of OS to boot, when you power on the system. GRUB is the boot loader used by most of the GNU/Linux systems.

Bibliography

[2]   http://en.wikipedia.org/wiki/List_of_curves

[3]   http://www.gap-system.org/~history/Curves/Curves.html

[4]   http://www.gap-system.org/~history/Curves/Ellipse.html

[5]   http://mathworld.wolfram.com/

[6]   http://www.scipy.org/Numpy_Example_List

[7]   http://docs.scipy.org/doc/

[8]   http://numericalmethods.eng.usf.edu/mws/gen/07int/index.html

[9]   http://www.angelfire.com/art2/fractals/lesson2.htm

[10] http://www.fizyka.umk.pl/nrbook/bookcpdf.html

[11] http://www.mathcs.emory.edu/ccs/ccs315/ccs315/ccs315.html

[12] Numerical Methods in Engineering with Python by Jaan Kiusalaas