An Introduction to Python and LaTeX

Release 0.01

Pramode C.E

February 18, 2010

CONTENTS

The BSc Informatics course conducted by University of Calicut is based completely on Free Software. Students are required to study the Python programming language and also learn how to prepare technical documents using LaTeX. Majority of the students have a computing background limited to using Microsoft Windows for simple tasks like web browsing. *An introduction to Python and LaTeX* has the objective of providing a gentle introduction to computing to such an audience.

Because most of the readers would be either Mathematics teachers or undergraduate Math students, sample programs are structured around simple mathematical concepts. No attempt has been made to go deep into language syntax - my experience with conducting several workshops for teachers has been that any attempt to introduce syntax elements whose application can't be demonstrated immediately through simple math examples usually leads to confusion. So, this book will not be an ideal resource for people who wish to master Python syntax and idioms in depth. There are plenty of great resources on the net for that, one of the best being *Dive into Python* (available at http://www.diveintopython.org).

This book is being updated continuously - the most recent version is openly available on the Internet at this address: http://radiantbytes.com/books/python-latex. Please visit this web site if you wish to contact the author or discuss problems encountered while teaching the subject.

CHAPTER

ONE

AN INTRODUCTION TO THE GNU/LINUX OPERATING SYSTEM

This book teaches programming using Python and technical document preparation using LaTeX. Both LaTeX and Python are Free Software, and the ideal way to learn them is by using the GNU/Linux operating system.

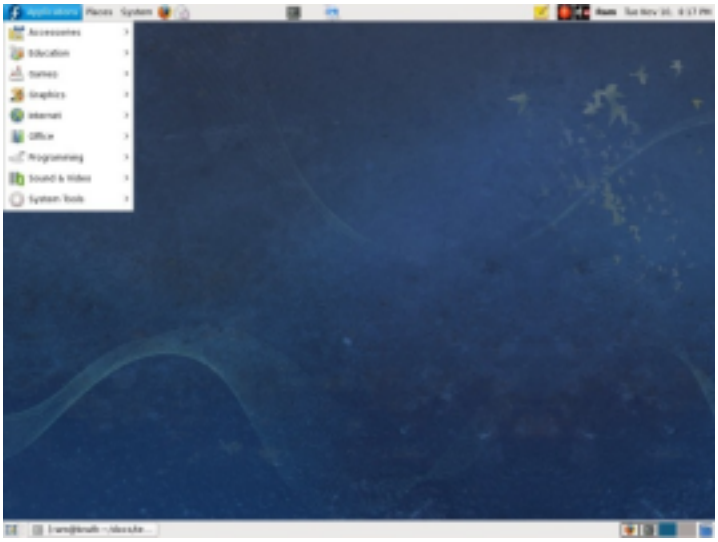
1.1 Installing GNU/Linux

A complete GNU/Linux system is composed of thousands of applications, all available for download on the Internet. Setting up a system by downloading and installing each component individually is a very tough job - so we have what are called *Linux distributions* which are simply CD's/DVD's packed with all the tools required to set up a fully functional system. There are several GNU/Linux distributions - Fedora, Ubuntu and Debian are some of the more popular ones. These distributions basically differ in the mix and match of software they provide - most people would advise beginners to get started with Ubuntu.

A complete Ubuntu distribution can be downloaded from the web site: http://www.ubuntu.com. The downloaded software should be written to a CD - this CD can be used to install Ubuntu on to the hard disk. The installation process is usually very easy and a person moderately experienced in using computers, even if he is using GNU/Linux based systems for the first time, should be able to complete it without too much trouble. We will not go into the details of system installation in this book - it is assumed that you have a working GNU/Linux system at hand.

1.2 Using GNU/Linux

Shown below is the screen shot of a Fedora GNU/Linux desktop.



You will find menus like *Applications*, *Places*, *System* etc. The menu layout and desktop appearance will be different on different GNU/Linux distributions. But that shouldn't bother you too much. A little experimentation is all that is required to get up and running fast!

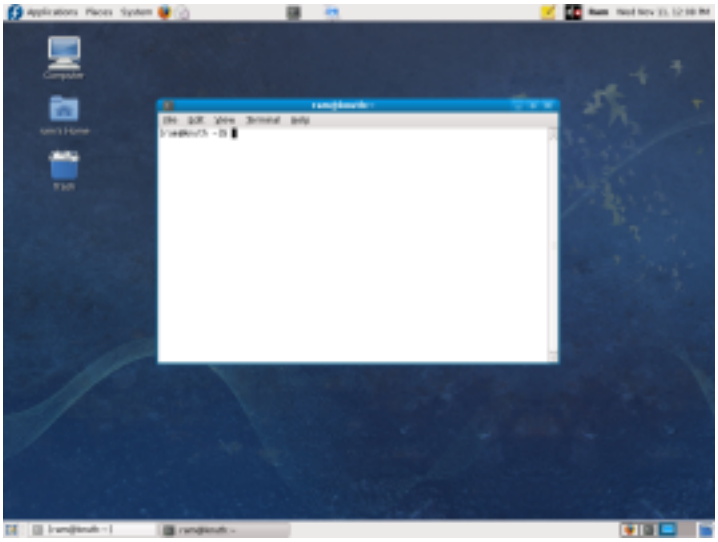
Here are some common GNU/Linux applications for tasks like editing documents, browsing the web etc.

- Firefox - the standard web browser on GNU/Linux systems
- OpenOffice - you can use this program for word processing, spreadsheet calculations.
- GIMP - a powerful image manipulation program. You will love this if you are a graphics artist.

1.3 The GNU/Linux Command line

The mouse based Graphical User Interface is ideal for beginners. But once you start writing programs you will have to interact with the operating system by typing commands. You need to open a terminal for doing this. On a Fedora system, the terminal is available in the *System tools* submenu of the *Applications* menu (its name is simply *Terminal*). The location of this program may be different on other GNU/Linux distributions - so you will definitely have to experiment a bit!

Here is a screen shot of a Fedora desktop with the terminal application active.



You can type commands within the terminal window and they would get executed. Say for example you wish to see the current date - the command for doing this is simply *date*. Here is what you might see if you type **date** (followed by the *Enter* key):

```
[ram@knuth ~]$ date
Wed Nov 11 12:28:19 IST 2009
[ram@knuth ~]$
```

The message:

```
[ram@knuth ~]$
```

which you see above is displayed automatically by the terminal - it is called a *command prompt*. The command prompt is simply a visual indication of the fact that the terminal is ready to accept commands.

Note: The actual message displayed as part of the command prompt will be different on your machine. No need to worry about it! Just keep in mind that you always type commands *after* the prompt.

In later chapters, you will have to use a few simple commands to interact with Python and LaTeX.

CHAPTER

TWO

GETTING STARTED WITH PYTHON

The Python programming language was created by a Dutch Computer Scientist called *Guido von Rossum*. Unlike languages like C/C++/Java, Python is extremely friendly to beginners - famous Universities like MIT (Massachusetts Institute of Technology) use it to teach introductory programming courses. It is also a modern, powerful general purpose programming language used by companies and research institutions all over the world to solve complex computing problems.

2.1 First steps - use Python as a calculator!

In the previous chapter, you had seen how to interact with the GNU/Linux Operating System by typing commands. You can run Python by simply typing the command **python**. Here is how your screen would look like:

```
[ram@knuth ~]$ python
Python 2.6 (r26:66714, Mar 17 2009, 11:44:21)
[GCC 4.4.0 20090313 (Red Hat 4.4.0-0.26)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Don't worry if the details shown above do not match exactly with what you see on your computer. It is enough that the last line of the output matches what you see above exactly. The three *greater than* signs constitute what is called the *python prompt*. When Python displays the prompt, you know that it is ready to accept commands! Type 1 + 2, hit the *Enter* key and see what happens:

```
>>> 1 + 2
3
>>>
```

Note: In examples like the one above, you should not type the three *greater than* signs - it is automatically displayed by Python.

Note: If Python shows you some error (instead of printing 3), read the section titled "use of space character in Python" which comes at the end of this chapter.

After you hit *Enter*, Python displays the result 3 on the next line and displays the *prompt* once again. This is Python's way of telling you that it is ready to accept more commands! Now you know that using Python is as easy as using a calculator.

Try a few more experiments:

```
>>> 3 * 2
6
>>> 3 - 2
1
>>> 3 / 2
1
>>> 3.0 / 2
1.5
>>> 2 ** 3
8
>>> 9 % 2
1
>>> 9 % 3
0
>>>
```

The *star* symbol performs multiplication and the *double star* performs exponentiation. The % symbol is the *remainder* operator. The only confusing result is the output from the division operation 3/2. This is explained in detail in the next section.

2.2 A few things about the way Python treats numbers

A number written like this:

```
456
```

is treated as an *integer* by Python. A number written like this:

```
456.01
```

is called a *floating point* number(or simply, a *float*). When division is performed, if both numerator and denominator are integers, the result will be truncated to an integer. So, when you divide 3 by 2, you get 1 and not 1.5. If you wish to get the correct answer, you have to make one (or both) of the numbers float.

Note: This behaviour (of division) is confusing and it has been corrected in a more recent version of Python.

You will encounter some other problems when dealing with floating point numbers; problems mostly concerning accuracy of the result. Here is a simple experiment you can try at the Python prompt:

```
>>> 1.5 + 1.1
2.6000000000000001
>>>
```

You may be surprised at not getting exactly 2.6 as the result. The reason for this behaviour is somewhat complex and its explanation is beyond the scope of this book. At this point, it is sufficient to understand that floating point arithmetic is tricky (in all programming languages, not just Python).

Note: Floating point numbers can be written in a different way - as an example, the number 0.00000001 may be written as 1e-8 (1 * 10 to the power of -8). Another example: 1234.0 can be written as 1.234e3 or 12.34e2 or 123.4e1.

2.3 Arithmetic using very large integers

Try the following experiment:

```
>>> 2 ** 320
21359870359209100823950217061695521146027045223566
52769947041607822219725780640550022962086936576L
>>>
```

Python can handle very large integer values. Note that the number shown above has an **L** at the end - this is Python's way of saying that you are dealing with a *long* integer! (Python does not do any kind of approximation when representing such large numbers - the number you have seen above is the *exact* value of 2 to the power of 320).

2.4 Using variables

Just like math, Python too has variables. The following example shows a simple use of variables:

```
>>> x = 1
>>> y = 2
>>> x + y
3
>>>
```

Here, x and y are variables having values 1 and 2. A variable should always have a value assigned to it before it is used. In the above example, Python will generate an error if you try to do x + y + z:

```
>>> x + y + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>>
```

Look at the last line - it says *NameError: name 'z' is not defined* - the problem is, you have not given a value to variable z (you need not bother with the other two lines in the error message - they can be safely ignored).

Mathematicians usually use single letter variable names like *x*, *y*, *z*, *a*, *b* etc. But Python has no problems with longer names:

```
>>> mark = 90
>>> age = 19
```

Here, *mark* and *age* are two variables just like *x* and *y* in the previous example. There are restrictions on the kind of words you can use as variables. For example, it's OK to use the name *twenty20* as a variable name, but you can't use the name *20twenty*. The rule is that variable names have to start with an uppercase or lowercase alphabet (an *underscore* symbol is also permitted, but digits like 0, 1, 2 etc are not).

Variable names are case-sensitive - for example, the names *Abc* and *abc* represent two different variables.

Once you assign a value to a variable, it remains unchanged as long as you do not assign a new value:

```
>>> x + y
3
>>> x = 10
>>> y = 20
>>> x + y
30
>>>
```

Initially, *x* and *y* have values 1 and 2 which are then changed to 10 and 20.

2.5 Mistakes made by beginners while using variables

- Not realizing that variable names are case-sensitive.
- If variable *x* has value 1, writing x + 1 does not change the value of *x* to 2:

```
>>> x = 1
>>> x + 1
2
>>> x
1
>>>
```

- If variable *x* has value 1, then writing x.1 does not give you 1.1. Variable names simply do not work that way!

The last mistake is not common, but I have seen students making it in the class!

2.6 Use of the space character in Python

Another common problem is regarding the use of space. If you wish Python to evaluate the expression 1+2, just type 1, followed by the '+' symbol, followed by 2 and then hit Enter. If you hit the spacebar key before typing 1, you will get an error which looks like this:

```
>>>  1 + 2
File "<stdin>", line 1
 1 + 2
 ^
IndentationError: unexpected indent
>>>
```

It's OK to type as many spaces as you wish between 1 and the '+' symbol and between the '+' symbol and 2, but if you type a space at the beginning of the line, that will create trouble. In a later chapter, you will see situations where you have to type one or more spaces at the beginning of a line.

2.7 Exiting Python

If you wish to stop using Python, simply type *Ctrl-d* (hold down the key labeled *Ctrl* on the keyboard and then type the letter **d**). All the variable assignments which you have made are lost when you exit Python.

Note: It is assumed that you are running Python on a GNU/Linux system. On a Windows system, you have to type *Ctrl-z* and then hit Enter.

2.8 Exercises

1. In mathematics, a *Mersenne number* is a positive integer that is one less than a power of 2.

$$M_p = 2^p - 1$$

   A *Mersenne prime* is a Mersenne number that is prime. As of October 2009, only 47 Mersenne primes have been discovered; the largest known prime number is a Mersenne prime.The 27th Mersenne prime (with 13395 digits in it) is obtained if you use 44497 as the value of *p* in the above equation. Use Python to find out the value of this number. (Interested students might wish to check out the *Great Internet Mersenne Prime search* on the net).

2. Find factorial of 20 using Python.

3. Wilson's theorem states that the number (p - 1)! + 1 is divisible by p for all primes p. Also, if p is not a prime number, then (p - 1)! + 1 is not divisible by p. Try a few experiments in Python to demonstrate this theorem.

4. Fermat's little theorem states that if *p* is a prime number, then for any integer *a*:

$$a^p - a$$

   is divisible by *p*. Use Python and test the theorem with some large prime numbers.

5. Find out what happens if you divide a number by zero in Python.

CHAPTER

THREE

USING PYTHON LISTS

We have seen how Python works with numerical data. Now, let's look at some of the techniques used by Python programmers to handle large collections of data.

3.1 Elementary list manipulation

3.1.1 Creating a list

A point in the Cartesian plane is represented by two numbers - its X and Y co-ordinates. Using simple Python variables, we might write:

```
>>> x = 2
>>> y = 3
>>>
```

But this does not in any way make it clear that we are referring to the x and y coordinates of a single point; we need some way to group together these two numbers. Python provides a powerful *list* notation to do this:

```
>>> p = [2,3]
>>>
```

The idea is simple - just write a sequence of numbers (separated by comma) within two brackets (note that we are using *square brackets* here) and you have what is called a Python *list*. In the above case, we can say that the variable *p* represents a point with co-ordinates 2 and 3.

The number of elements in a list is not limited - you can have as many as you like:

```
>>> marks = [80, 75, 66, 55, 95]
>>>
```

In this case, *marks* represents the mark of five students in an exam.

3.1.2 Indexing a list

The variable *marks* represents a set of marks *as a whole*. Suppose you wish to know the mark of each student individually. Here is how it is done:

```
>>> marks
[80, 75, 66, 55, 95]
>>> marks[0]
80
>>> marks[1]
75
>>> marks[4]
95
>>>
```

When you write marks[0], you are asking Python to display the first mark in the sequence and when you write marks[4], you are asking Python to display the fifth mark in the sequence. Note that Python is counting from zero, that is why the first mark is marks[0] and not marks[1].

You can read the statement marks[0] as either *marks zero* or *marks of zero*. The operation of looking at individual elements of a list is called *indexing*.

What will happen if you write *marks[5]*? Let's find out:

```
>>> marks[5]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

Look at the last line of the error message; it says *IndexError: list index out of range*. The elements of the list are numbered from 0 to 4 - you are asking Python to display an element in the sequence which is simply not present and so you get an error!

3.1.3 Modifying the elements of a list

Let's say you wish to change the mark of the first student from 80 to 65. Here is how it is done:

```
>>> marks[0] = 65
>>> marks
[65, 75, 66, 55, 95]
>>>
```

3.1.4 Appending elements to a list

How do you append a new mark to the *marks* list?:

```
>>> marks.append(41)
>>> marks
[65, 75, 66, 55, 95, 41]
>>> marks.append(15)
>>> marks
[65, 75, 66, 55, 95, 41, 15]
>>>
```

3.1.5 Deleting an element from a list

Say you wish to delete the third element of the *marks* list:

```
>>> del marks[2]
>>> marks
[65, 75, 55, 95, 41]
>>>
```

CHAPTER

FOUR

INTRODUCTION TO NUMERIC ARRAYS

Let's try an experiment with lists:

```
>>> a = [1, 2, 3, 4]
>>> a + 1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>>
```

Our idea is to check whether it is possible to add 1 to each element of the list by simply writing:

```
a + 1
```

Unfortunately, Python does not let us do this (there is very good reason as to why the language behaves in this way). But that does not mean that there is no way to do arithmetic on a sequence of numbers; Python offers powerful *Numeric Arrays* using which we can do all kinds of tricky number sequence manipulations. But before we use Numeric Arrays, we have to understand how to use *modules* in Python.

4.1 Using Modules in Python

Try this experiment:

```
>>> sin(0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>>
```

We are unable to calculate sin of 0 - Python tells us that *name 'sin' is not defined*! The reason is simple; functions like *sin*, *cos* etc are not part of the *core* Python language. They are defined in

additional *modules* (also called *libraries*). We have to use a special notation to tell Python that we wish to use the functions defined in a module:

```
>>> from math import *
>>> sin(0)
0.0
>>> cos(0)
1.0
>>>
```

Note: Functions like sin, cos etc need angles to be specified in radian.

The first line:

```
from math import *
```

simply means:

```
Use all the functions available in a module called 'math'.
```

The *star* symbol should not be interpreted as a multiplication operator in this context - it simply means *ALL*.

In general, if you wish to use all the functions defined in a module called *xyz*, just type:

```
from xyz import *
```

4.1.1 Another way to use modules

One of the things which makes programming languages tricky to learn is that there will be many different ways to express the same idea. Here is another way to use a module:

```
>>> import math
>>> sin(0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> math.sin(0)
0.0
>>> math.cos(0)
1.0
>>>
```

The statement:

```
import math
```

is similar to the one which we saw in the previous section - the difference is that now we have to write *math.sin* and *math.cos* instead of simply *sin* and *cos*.

What do you do if you wish to get a list of all the functions present in a module?:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees',
'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

The function *dir* can be used to display list of all functions defined in a module. You can ignore the names starting with two *underscore* signs in the above list.

The working of many of these functions should be evident from their names. If you wish, you can get information about what the function actually does by using *help*:

```
>>> help(math.hypot)
hypot(...)
 hypot(x,y)

 Return the Euclidean distance, sqrt(x*x + y*y).
>>>
```

4.2 Creating Numeric Arrays

A Python *Numeric Array* looks like a list - with the difference that it is more suited for mathematical computations. Here is how you can create a Numeric array:

```
>>> from pylab import *
>>> a = [10, 20, 30, 40]
>>> b = array(a)
>>> b
array([10, 20, 30, 40])
>>> b + 1
array([11, 21, 31, 41])
>>> b * 2
```

```
array([20, 40, 60, 80])
>>>
```

The first statement:

```
from pylab import *
```

simply makes available all the functions in a module called *pylab*. If you get an error while executing this statement, it means the module is not installed on your system. Read the Appendix of this book to learn how you can install additional software on your GNU/Linux system.

We are making use of only one function from the *pylab* module in the above example; a function called *array*. What does *array* do? It acts on an ordinary Python list *a* and produces a *Numeric sequence* called (in this example) *b*. You can now add 1 to all the elements of the sequence by just writing:

```
b + 1
```

Similarly, you can multiply all the elements of *b* with 2 by doing:

```
b * 2
```

Now, just relax for a moment and think of the power that you have in your hands! A numeric array (like *b* above) can contain hundreds of thousands of numbers. With a simple statement like "b+1", you can operate on all the elements of the sequence! Working with a million numbers at once has become as easy as working with one or two numbers.

You will see how to use this power to do some interesting math in the next chapter!

CHAPTER

FIVE

DOING MATH WITH NUMERIC ARRAYS

In this chapter, we will see how to plot curves using Numeric arrays and functions like *plot* and *polar*.

5.1 Plotting a sine curve

Say you wish to plot one full cycle of the sine curve. The first step is to generate a set of numbers between 0 and 2*pi. This can be done very easily using a function called *linspace*:

```
>>> from pylab import *
>>> a = linspace(0, 2*pi, 4)
>>>> a
array([ 0.        ,  2.0943951 ,  4.1887902 ,  6.28318531])
>>> sin(a)
array([  0.00000000e+00,   8.66025404e-01,  -8.66025404e-01,
     -2.44921271e-16])
>>>
```

The function:

```
linspace(x, y, N)
```

generates N equally spaced numbers starting from *x* and ending in *y*; the difference between adjacent numbers is fixed. In the above case, *x* is 0, *y* is 6.28 and *N* is 4. The output which *linspace* generates is a Numeric Array:

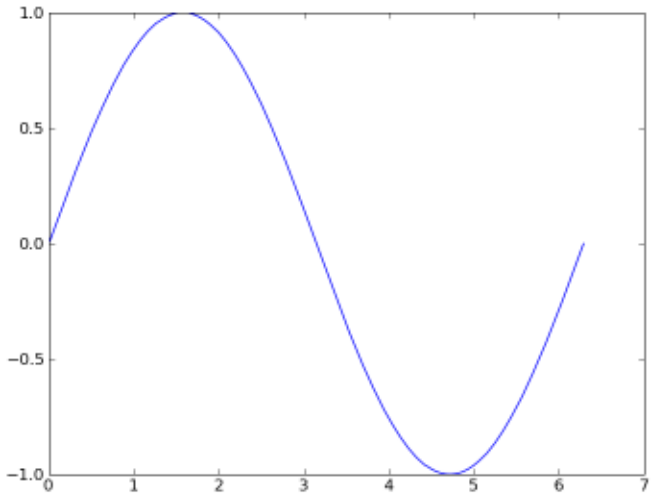
array([0, 2.0943951 , 4.1887902, 6.28318531])

You observe that the difference between consecutive numbers is fixed, approximately equal to 2.09. When you apply the *sin* function on *a*, it computes sine of *all* the numbers in the numeric sequence.

Plotting a sine curve with just 4 points is not a good idea; especially when it is so easy to generate any number of points:

```
>>> x = linspace(0, 2*pi, 200)
>>> y = sin(x)
>>> plot(x, y)
[<matplotlib.lines.Line2D object at 0x9bf4bac>]
>>> show()
>>>
```

Now, we are generating 200 points (in x) and storing the *sin* of each of these points in y. The *plot* function works like this: it takes the first number in x and the first number in y and plots a point, then it takes the second number in x and the second number in y and plots another point ... and so on. Note that when you apply *plot*, it does not immediately show you the graph - it just prints a line in the output (we need not understand what that line means). Only when the *show* function is applied do we actually get the plot on the screen!



5.2 Plotting the polar rose

In mathematics, the polar coordinate system is a two-dimensional coordinate system in which each point in a plane is determined by a pair of values:

$$(r, \theta)$$

where r represents the distance from a fixed point (called the *pole*) and *theta* represents an angle with respect to a fixed direction.

It is easy to plot in polar coordinates, as the following example illustrates.

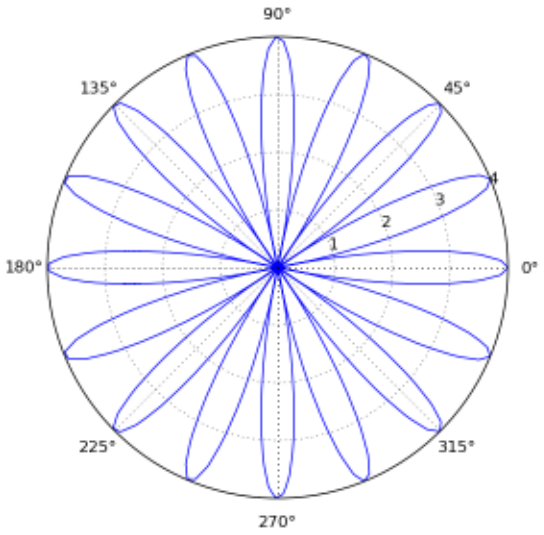
The *polar rose* is defined as:

$$r(\theta) = acos(k\theta)$$

We get a k-petaled rose if *k* is odd and *2k* petaled rose if *k* is even. Here is a Python code fragment which plots the polar rose:

```
>>> from pylab import *
>>> theta = linspace(0, 2*pi, 200)
>>> r = 4 * cos(8 * theta)
>>> polar(theta, r)
[<matplotlib.lines.Line2D object at 0xa3efb4c>]
>>> show()
>>>
```

The function *polar* works with two numeric arrays - the second one gives the *r* values of each point in the plane and the first one gives the corresponding angles. Here is the plot:



5.3 Labels and title

You can give a title to your plot by using a function called *title* (call this function before calling *show*):

```
>>> title('A simple Sin curve')
<matplotlib.text.Text object at 0x8d31bec>
>>>show()
```

Labels can be given to the X and Y axes:

```
>>> xlabel('X axis')
<matplotlib.text.Text object at 0x89d26ec>
>>> ylabel('Y axis')
<matplotlib.text.Text object at 0x8a87b8c>
>>> show()
```

xticks and *yticks* are two other interesting functions. Try out the experiment below to understand how they work:

```
>>> x = linspace(0, 2*pi, 200)
>>> y = sin(x)
>>> m = arange(0, 2*pi, .4)
>>> n = arange(-1, 1, .2)
>>> xticks(m)
>>> yticks(n)
>>> plot(x, y)
>>> show()
```

Note: The function call *arange(0, 2*pi, 0.4)* returns a numeric sequence from 0 to 2*pi - each number in the sequence differs from the next by 0.4.

5.4 Exercises

1. Refer your maths textbook and find out the polar equation of the curve called a *cardioid*. Try to plot the *cardioid* using Python.
2. Try to plot a circle in polar coordinates.

CHAPTER

SIX

DEFINING FUNCTIONS

So far, we have been making use of functions (like *sin*, *linspace*) etc provided by Python. In this chapter, we shall see how to create our own functions.

6.1 Function basics

Try the following experiment at the Python prompt:

```
>>> from math import *
>>> sqrt(16)
4.0
>>> sqr(4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'sqr' is not defined
>>>
```

Note: from math import * is required because function *sqrt* is defined in the math library.

Python tells us that the name *sqr* is not defined; there is a function defined in the math library for computing the square root, but there is no function for squaring a number! Not just Python, but any programming language will provide you with only a limited set of functions; if you wish, you can create your own functions!

You can define a *sqr* function of your own:

```
>>> def sqr(x):
...     return x * x
...
>>>
```

A simple function works like this - you give it one or more numbers, it performs some calculations using these numbers and gives back the result. In the above case, you are asking Python to:

```
Define a function which takes in one number x and gives back x*x.
```

It is as simple as that; but you will have to follow a few syntax rules strictly if you want your function definitions to work properly.

The first line of a new function definition starts out like this:

```
def functionname ( variablename ) :
```

Here, *def* is a language keyword, which simply means *define a new function*. This should be followed by the name of the function you are creating (in this case, *sqr*. There is nothing special about function names, they are just like variable names - instead of *sqr*, you could have as well used a name like *xyz* and Python will not bother about it), followed by a bracket, followed by one (or more) variable names, followed by a closing bracket, followed by a *colon* symbol.

The remaining lines that you type forms the *body* of the function (the *body* is where the function performs all of its computations) - in the above case the body is composed of only one line:

```
return x * x
```

The first thing that you note is that instead of the usual three greater-than signs, you are seeing:

```
...
```

This is Python's way of telling you that what you are typing right now is part of the *body* of the function. You also note that the line "return x * x" is *indented* (that is, there are a few spaces between the ... and "return x * x") - this is again part of Python's syntax - the lines in the body of a function have to be uniformly indented (you can do it by typing a few spaces, or more conveniently, by simply typing a *tab*. Never mix spaces and tabs - that is a syntax error in Python).

Now, what does the *return* statement do? For the moment, just think of it as another tiny bit of syntax!

How does Python know that you have finished typing the body of the function? When we type an empty line! At this point, Python prints the original "three greater-than sign" prompt.

Once you have defined *sqr*, you can use it just like all the the other functions:

```
>>> sqr(4)
16
>>> sqr(3)
9
>>> m = sqr(5)
>>> m
25
>>>
```

6.1.1 A few technical terms

In programming language terminology, a statement like this:

```
sqr(3)
```

is called a:

```
function call or function invocation
```

When you write:

```
>>> def sqr(x):
```

the *x* within the brackets is called a function *parameter* or *argument*. The *argument* is always a variable. A function can have more than one parameter. Here is a simple example:

```
>>> def add(x, y, z):
...     return x + y + z
...
>>>
```

You can call the above function in many ways:

```
>>> add(1, 2, 3)
6
>>> p = 1
>>> q = 2
>>> r = 3
>>> add(p, q, r)
6
>>>
```

In the first case, when you call:

```
add(1, 2, 3)
```

the parameter *x* in *add* will assume the value 1; the parameters *y* and *z* will assume values 2 and 3.

In the second case, when you call:

```
add(p, q, r)
```

the parameter *x* will assume value of variable *p*; parameters *y* and *z* will assume values of variables *q* and *r*.

What will happen if you try something like this:

```
>>> add(1, 2)
```

Python will give you an error - this makes sense because in this case, you are not supplying a value for parameter *z*.

Similar is the case if you try calling like this:

```
>>> add(1, 2, 3, 4)
```

Now, you are trying to call the functions with more values than are required. This also is a serious error.

6.1.2 Functions with multiple statements in the body

Length of a line segment connecting the points (a, b) and (c, d) is:

$$l = \sqrt{(c-a)^2 + (d-b)^2)}$$

Let's define a Python function *line_length* to compute this length:

```
>>> from math import *
>>> def line_length(a, b, c, d):
...       return sqrt((c -a) * (c - a) + (d - b) * (d - b))
...
>>>
```

Note: You should give meaningful names to your functions. In the above case, the function is called line_length - in case your function name has multiple words in it, it's a good idea to separate the words using an *underscore* symbol, _. Python does not care even if you call your function *linelength* - the underscore simply improves readability of your code.

Instead of writing a complicated expression in a single line, let's do it in a different way:

```
>>> def line_length(a, b, c, d):
...     m = (c - a) * (c - a)
...     n = (d - b) * (d - b)
...     return sqrt(m + n)
...
>>>
```

Note: Remember, you have to type an empty line after the last line in the body of the function to tell Python that you have finished typing the body.

Both the functions perform the same action - but in the second case, we are simplifying things a bit by using two extra variables *m* and *n* and two additional lines in the body. The important thing to

be kept in mind here is that all the three lines in the body have to be at the same level of indentation - that is, if the first line in the body is separated from the "..." by say four spaces, then the two other lines too need to have exactly the same separation. An easy way to achieve uniform spacing is to use the *tab* key instead of the *space bar* on your keyboard.

Note: You should not mix tabs and spaces - either use tabs uniformly or use space uniformly.

Here is one more way to write the above function:

```
>>> def line_length(a, b, c, d):
...       m = sqr(c - a)
...       n = sqr(d - b)
...       return sqrt(m + n)
...
>>>
```

We can get rid of the two extra variables and write:

```
>>> def line_length(a, b, c, d):
...       return sqrt(sqr(c-a) + sqr(d - b))
...
>>>
```

You have to define the *sqr* function properly if both the above examples are to work. The idea being conveyed here is that you can build more complex functions using simpler functions - we used both *sqrt* and *sqr* to write *line_length* (maybe, you can imagine an even more complex function which uses *line_length* in its body)!

6.2 What does 'return' do?

It's time for us to examine the significance of *return*. Let's try to rewrite *sqr*:

```
>>> def sqr(x)
...     x * x
...
>>>
```

Now, let's call it:

```
>>> sqr(4)
>>> m = sqr(3)
>>> m
>>>
```

You note that nothing is getting printed. Even though Python performs the computation 'x*x', the result of the computation is made available to you only if there is a *return* statement. The function is next to useless without the *return*.

There is another aspect of *return* which the following function illustrates:

```
>>> def silly(a):
...     m = a + 1
...     return m
...     n = a - 1
...     return n
...
>>> silly(10)
11
>>> silly(23)
24
>>>
```

We have used two return statements in the body of the function. The result of calling silly(10) is 11 and silly(23) is 24; this means that the two lines written after the *return m* have no effect at all. So, the big idea here is:

```
Your function effectively stops executing after the first return.
```

6.3 Boolean Functions

Try the following experiments at the Python prompt:

```
>>> 1 > 2
False
>>> 1 < 2
True
>>> 1 == 1
True
>>> 1 <> 2
True
>>> (8 % 2) == 0
True
>>> not True
False
>>> not False
True
>>>
```

The symbols *True* and *False* (in computing terminology, you call them *boolean* values) are used by Python to identify whether the result of a comparison operation is true or false.

The operator used to check whether two values are equal is '==' . Note that this is totally different from the '=' operator (the *assignment* operator) which is used to give value to a variable. The '<>' operator returns True if the numbers being compared are not equal.

In the expression:

```
(8 % 2) == 0
```

% is the remainder operator; remainder when 8 is divided by 2 is zero. So, the above expression becomes:

```
0 == 0
```

which is True.

What does the following function do?:

```
>>> def is_even(n):
...       return (n % 2) == 0
...
>>> is_even(4)
True
>>> is_even(5)
False
>>>
```

is_even divides its parameter *n* by 2 and finds out the remainder which is either zero or one; zero if number is even and one if number is odd. If, for example, n has the value 4, the expression:

```
(n % 2) == 0
```

becomes:

```
0 == 0
```

which is true! Effectively, *is_even* checks whether its parameter *n* is even. A function like *is_even* which returns True or False is called a *boolean* function.

6.4 Using if-else statements

Look at the following function definition:

```
>>> def maximum(a, b):
...     if (a > b):
...         return a
...     else:
...         return b
...
>>> maximum(1, 2)
2
>>> maximum(20, 10)
20
>>>
```

You can see two new Python keywords here - *if* and *else*; together, they form an *if-else* statement. An *if-else* statement is merely Python's way of saying:

```
if this condition is true, do this action; otherwise, do some other action.
```

The general format of an *if-else* statement is:

```
if (condition):
    body
else:
    body
```

The *body* can be either a single statement or it can be multiple statements written one below the other. Special care should be taken to make sure that the statement(s) in the body are uniformly *indented* (using equal number of spaces or a single tab). In the case of our *maximum* function, because the *if-else* itself comes as the body of a function, we have two levels of indentation. Here is how you should type the code:

```
Type the first line - def maximum(a, b): and hit Enter
Type a tab and then type - if (a > b): and hit Enter
Type two tabs and then type - return a  and hit Enter.
Type a single tab and then type - else: and hit Enter
Type two tabs and then type - return b and hit Enter
Type one more Enter to insert a blank line
```

You need to type two tabs before typing "return a" and "return b" - this is because Python uses indentation levels to decide whether a statement comes as the body of an *if* (or *else*) or a function definition. Only if the line "return a" is indented deeper than the line "if (a>b):" will Python assume that the "return a" is part of the body of the *if* statement. Getting the indentation levels wrong is a common mistake made by beginners!

6.5 Exercises

1. Define a function is_odd which checks whether a number is odd or not.
2. Define a function maximum which returns the maximum of three numbers (Hint: you might first define a function which will find out maximum of two numbers and use that to write a function which will return maximum of three numbers).

CHAPTER

SEVEN

FUNCTIONS WITH SIDE EFFECTS

Python functions are almost like mathematical functions - you give the function some input (give the number 2 to the *sqr* function), it performs some computation and gives you an output (4, in the case of the *sqr* function). In fact, the only reason we are using a function is because we are interested in getting its output.

But this need not be the case always. In this chapter, we examine how to write functions which have *side effects*.

7.1 Modeling a bank account

Most useful computer programs are closely connected with the real world. When you go to a bank to deposit or withdraw money, you are interacting with a computer program which knows everything about the accounting principles followed by that bank. Here is a toy Python program which tries to do a bit of *banking*!

```
>>>balance = 1000
>>>def deposit(amount):
...     global balance
...     balance = balance + amount
...
>>>
>>>def withdraw(amount):
...     global balance
...     balance = balance - amount
...
>>>
```

We first create a variable called *balance* initialized with the value 1000 (say we have 1000 Rupees balance initially). Then we define two simple functions *deposit* and *withdraw*. The interesting things about these functions are:

- Both functions do not use *return*

- There is a new keyword in the body of both functions - *global*

The line:

```
global balance
```

simply tells Python that the variable *balance* being used in the next line (and all the other lines of the function, if any) refers to the variable *balance* declared outside the function (such variables are called *global variables* in programming language terminology). Let's see what happens when we call this function:

```
>>> balance
1000
>>> deposit(100)
>>> balance
1100
>>> deposit(30)
>>> balance
1130
>>>
```

Initially, the variable *balance* has value 1000. When you compare the call:

```
>>> deposit(100)
>>>
```

with the call:

```
>>> sqr(2)
4
>>>
```

one thing becomes clear - *deposit* is not *returning* any value, unlike *sqr* which gives back 4. This is because we simply have not used a *return* statement in the body of *deposit*. Still, *deposit* does something useful; it adds 100 to the global variable *balance*. This is evident when we ask Python to display the value of *balance* - indeed, it has changed to 1100. Yet another call to *deposit* with argument 30 results in 30 being added to the balance.

Similar is the case when you withdraw some money from our toy Python bank:

```
>>> balance
1130
>>> withdraw(20)
1110
>>> withdraw(10)
1100
>>>
```

The two functions above (*deposit* and *withdraw*) are said to have a *side effect* - in this case, the *side effect* is altering the value of a global variable. Both these functions are useful precisely because they have this *side effect*.

If you go deeper into programming, you will see many situations where you will define functions mostly to generate some kind of side effect, rather than to evaluate some mathematical expressions. Those topics are beyond the scope of this book.

CHAPTER

EIGHT

UNDERSTANDING THE WHILE LOOP

A set of Python statements can be executed repeatedly using a *while* loop. This chapter examines the syntax of the while loop using some simple examples.

8.1 A simple while loop

Type the following code at the Python prompt:

```
>>> while (1 < 2):
...     print 'hello'
...     print 'world'
...
```

The Python *print* statement can be used for displaying a message on the screen; the message should be in single (or double) quotations. Both the print statements should be indented properly. You should type an empty line after the last print statement. You will see that Python keeps on printing the two messages *hello* and *world* repeatedly!

The general format of a *while* loop is:

```
while (condition):
   body
```

The *body* can be either a single statement or multiple statements, all properly indented (just like the body of a function or an if-else statement). Python first checks the condition and if it is found to be true, the body gets executed. Once again, Python checks the condition and if it is found to be true, the body is executed again. This process repeats until the condition becomes false. If the condition never becomes false, Python will keep on repeating this process infinitely. In the above example, the condition is:

```
1 < 2
```

and the body is composed of two statements:

```
print 'hello'
print 'world'
```

Python first checks the condition - 1 is definitely less than 2, so the condition is true and the body (the two print statements) gets executed. Python once again checks the condition - again, 1 is less than 2, condition is true and so the body gets executed. 1 is never going to be bigger than 2, so the condition will never become false and as a result, Python will never stop executing the loop! If you wish, you can type *Ctrl-c* (hold down the *Ctrl* key and press **c**) to forcibly terminate the loop.

8.2 A loop which terminates

Here is a *while* loop which does something a bit more interesting than printing *hello* and *world*:

```
>>> i = 0
>>> while (i < 5):
...       print i
...       i = i + 1
...
```

The condition is initially true - i has the value 0 and zero is less than 5. So the body gets executed. The *print* statement displays the value of *i* on the screen. The next statement is:

```
i = i + 1
```

You should read this as: *i assigned to i plus 1*. The logic is:

```
New value of i is old value of i plus one.
```

So, the value of *i* becomes 1. Python once again checks the condition - it's true (because 1 is less than 5). So the body gets executed again. The value of *i* is printed(this time, the value is 1). The next statement(i = i + 1) changes the value of *i* to 2. The condition gets checked once more. Again, it is true (2 is less than 5). So the body gets executed once more. The print statement displays 2 on the screen (value of *i* is now 2) and the value of *i* gets changed to 3 in the next statement. This process repeats until the value of *i* becomes 5. At this point, the condition becomes false and the loop terminates. Here is what you will see on the screen:

```
0
1
2
3
4
```

8.3 Finding HCF using Euclid's method

The term *algorithm* is commonly used in computer programming. It refers to a precise set of rules for solving a problem. A very famous numerical algorithm exists to find out the highest common factor (also called the greatest common divisor) of two numbers. The algorithm was invented by Euclid; it's therefore called the *Euclid's algorithm*.

Here is how the algorithm works. Say you wish to find out the HCF of two numbers A and B. Simply repeat the following steps until both numbers become equal:

- If A is greater than B, change A to A-B and do not modify B
- If B is greater than A, change B to B-A and do not modify A

The value of A (and B) when both become equal will be the Highest Common Factor!

Let's try this out with two numbers, A=18 and B=42.

B is greater than A, so B becomes 24 and A remains as 18. So, we have:

```
A = 18, B = 24
```

If we repeat this process, we get:

```
A = 18, B = 6
A = 12, B = 6
A = 6,  B = 6
```

Finally, both A and B become equal (both have value 6). The HCF of 18 and 42 is 6.

You can try out this procedure on different pairs of numbers and verify that it works perfectly!

Even better, you can write a loop in Python to discover the HCF:

```
>>> a = 18
>>> b = 42
>>> while (a <> b):
...     if (a > b):
...         a = a - b
...     else:
...         b = b - a
...
>>> a
6
>>> b
6
>>>
```

Initially, *a* and *b* are not equal, so the condition "a <> b" (remember, '<>' is the *not equal to* operator) is true. As *b* is greater then *a*, the *else* part of the *if-else* statement gets executed and value of *b* changes to 24 while *a* remains unchanged. In the next iteration of the loop, *b* becomes 6 and *a* remains as 18. This process repeats until both *a* and *b* become equal. At that point, the loop terminates. Now, if you examine the value of *a* and *b*, you will see that both are 6.

A small problem with this approach is that any time you wish to calculate the HCF of two numbers, you have type the whole loop all over again. This is difficult. The next section shows how this problem can be solved.

8.4 Defining a function to compute HCF of two numbers

Look at the following Python program:

```
>>> def hcf(a, b):
...     while (a <> b):
...         if (a > b):
...             a = a - b
...         else:
...             b = b - a
...     return a
...
>>> hcf(18, 42)
6
>>> hcf(15,20)
5
>>>
```

The only change is that we are now writing the loop within a function called *hcf* (the name of the function doesn't really matter. You can call it by any name you like). Now whenever you wish to find out hcf of two numbers, you just have to call the function *hcf* with those two numbers as parameters. Say you call:

```
hcf(15,20)
```

Within the function, *a* will assume the value 15 and *b* will assume the value 20. The while loop will terminate when *a* and *b* both become equal to 5. Once the while loop is over, the *return* statement will transmit the value 5 as value of the function *hcf(15,20)*.

CHAPTER

NINE

FINDING SQUARE ROOT BY NEWTON'S ITERATION

We have used Python to write tiny programs, programs which are less than ten lines of code. Most useful programs are much longer; for example, the core of the Linux operating system (called the *kernel*) has a few million lines of code written in the C programming language! Such a system is the collective effort of hundreds of programmers. These programmers spend their time reading code written by others, making improvements and correcting errors. As a programmer, it is your responsibility to write clear, readable code. In this chapter, we examine how to use functions to achieve this objective.

9.1 Programming as Essay writing

How do you go about writing an essay (or a book, newspaper article, whatever)? First, you form an idea in your mind as to what exactly you wish to convey to your audience. The message you wish to convey should be presented in a structured manner; you will divide your essay into sections, sub-sections and paragraphs. You will make sure that ideas flow logically from one section to the next. If you are trying to explain something very complex, you will first introduce your audience to simpler (but related) ideas and then try to build up the more complex idea on the basis of the simpler ones. You will uphold clarity as one of your biggest virtues.

Writing a program is similar to composing an essay. Ultimately, your program has to run on a computer and provide correct output. But it should also be easy for other people to read, understand and modify your code. Let's study the Python implementation of a simple square root finding algorithm to understand how functions can be used to write clear code (the code I have presented here is the Python version of a program presented in the famous book *Structure and Interpretation of Computer Programs*).

9.2 Finding square roots

There is a very simple technique for calculating the square root of a number based on what is called the *Newton's method*. Let's say you wish to calculate square root of 16 (we will call this X). We make a guess that the root is 1:

```
X = 16, Guess = 1.0
```

Now, is this guess good enough? We can find out by taking the square of Guess and checking whether it is close to X. In this case, our guess is not good enough.

So, we will improve our initial guess; our "improved" guess should be, according to *Newton's method*:

```
Average of two numbers: Guess and X/Guess
```

In this case, it will be average of 1 and 16, that is 8.5.

This improved guess is also not good enough, so we shall improve it by again taking average of Guess and X/Guess where Guess is now 8.5. The value we get will be:

```
(8.5 + 16/8.5) / 2 = 5.19
```

We repeat this process a few times:

```
(5.19 + 16/5.19) / 2 = 4.13
(4.13 + 16/4.13) / 2 = 4.00
```

In just two more steps, we reach close enough to the required value!

9.3 Writing a square root finding program

In describing the root finding algorithm, we used terms like "improve", "average", "good enough" etc. The Python programming language has no idea what these terms stand for. But we can "teach" Python the meaning of these words by defining functions whose names are *improve*, *average* and *good_enough*. Let's give it a try.

9.3.1 Defining *average*

Here is our function *average*:

```
>>> def average(a, b):
...     return (a + b) / 2.0
...
>>> average(1, 2)
1.5
>>>
```

9.3.2 Defining *improve*

An *improved* guess is average of Guess and X/Guess. So, we define *improve* as:

```
>>> def improve(guess, x):
...      return average(guess, x/guess)
...
>>> improve(1.0, 16)
8.5
>>>
```

We give the names *guess* and *x* to the parameters of *improve*; we could have as well used any other names, say, *a* and *b*. The use of the names *guess* and *x* makes the code more readable.

Note the way we are using one function to build another (*improve* makes use of *average*).

9.3.3 Defining good_enough

We can say that *p* is square root of *q* if the difference between square of *p* and *q* is less than a small number say 0.001. We use this idea to define a *boolean function* called *good_enough* which checks whether its parameter *guess* is good enough to be the square root of *x*:

```
>>> from math import *
>>> def good_enough(guess, x):
...     d = abs(guess*guess - x)
...     return (d < 0.001)
...
>>> good_enough(1, 16)
False
>>> good_enough(4.0, 16)
True
>>>
```

The *import* statement is required because we are using a function called *abs* which is defined in the *math* library. *abs* returns the absolute value of its argument:

```
>>> abs(1)
1
>>> abs(-1)
1
>>>
```

We need to take the absolute value of the difference between guess*guess and *x* because guess*guess may be either greater than *x* or less than *x*; unless the absolute value is taken, this will cause problem when comparing with 0.001 in the next line.

The statement:

```
(d < 0.001)
```

is either True or False depending on whether *d* is less than or greater than 0.001.

9.3.4 Defining the square root function

The square root algorithm is very simple; basically, it says:

```
As long as the guess is not good enough, keep on improving the guess
```

Here is a function which implements this algorithm:

```
>>> def square_root(guess, x):
...     while(not good_enough(guess, x)):
...         guess = improve(guess, x)
...     return guess
...
>>> square_root(1, 16)
4.0000006366929393
>>> square_root(1, 25)
5.000023178253949
>>>
```

For finding square root of 16, we will call our function like this:

```
square_root(1, 16)
```

This will result in the parameter *guess* getting the value 1 and parameter *x* getting the value 16. In the condition part of the while loop, we have written:

```
not good_enough(guess, x)
```

Python calls the function *good_enough* with parameters 1 and 16 - *good_enough* will return False. The "not" operator, when it acts on the boolean value False, returns the value True. So we have the boolean value True in the condition part of the while loop. As condition is true, the body will get executed. The body is:

```
guess = improve(guess, x)
```

The *improve* function, given the values 1 and 16, returns 8.5. This becomes the new value of *guess*. The condition checking part of the while loop gets executed again; function *good_enough* gets called with parameters 8.5 and 16 and it returns False. The not operator returns True and so the body of the loop gets executed once more. This process stops only when *good_enough* returns True (in which case the condition becomes False because not True is False). If:

```
good_enough(guess, x)
```

is True, that means *guess* may be taken to be square root of x. When the loop terminates, the next statement:

```
return guess
```

returns the value of guess as the value of the function square_root(1, 16).

The only problem with our *square_root* function is that we have to call it with two parameters, the first one being the initial value for "guess". It is more natural to have a square root function which takes only one parameter - the number whose square root is required. We can manage this very easily by defining an additional function:

```
>>> def my_sqrt(x):
...     r = square_root(1, x)
...     return r
...
>>>
```

9.4 Conclusion

We developed our square root function on top of other, simpler functions. Each function we wrote performed one simple computation and was easy to read and understand. The final *square_root* function too was very simple (all the hard work was being done by the other functions). Finding the square root is definitely not a big deal, and real life programs are incredibly more complex than this toy example. But the approach we have taken scales well - most big programs are written this way, as a collection of simple functions.

CHAPTER

TEN

STRINGS, TUPLES AND ASSOCIATIVE ARRAYS

10.1 What is a string?

Let's say you are writing a program to handle information regarding students in your school/college. You can represent a student in a Python program as a simple list containing attributes like the student's name, age, marks etc:

```
>>> a = [Rahul, 18, 90]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'Rahul' is not defined
>>>
```

Python shows you an error! The reason is Python treats the name *Rahul* as a variable name; we have not defined a variable called Rahul and so the error.

What we want to do is tell Python to treat *Rahul* not as a variable but as a simple sequence of alphabets - this can be done by putting the name within single (or double) quotations. Such sequences are called *strings* in programming language terminology:

```
>>> a = ["Rahul", 18, 90]
>>>
```

10.1.1 Elementary string handling

Try the following sequence of operations at the Python prompt:

```
>>> a = "hello"
>>> b = "world"
>>> a + b
```

```
'helloworld'
>>> a[0]
'h'
>>> a[4]
'e'
>>> a * 3
'hellohellohello'
>>> len(a)
5
>>> a == b
False
>>>
```

In the first two lines, we are creating two variables, *a* and *b* referring to two strings *hello* and *world*. The addition operator can be applied on strings - it performs string concatenation (joins the strings together). Just like lists, strings too can be indexed; *a[0]* refers to the first element in the sequence. The multiplication operator performs repeated concatenation. It's possible to compare one string with another using the == operator; in the above case, the comparison gives us the result *False*.

10.2 What is a tuple?

A Python *tuple* looks similar to a list, but has some fundamental differences:

```
>>> a = (1, 2, 3)
>>> a
(1, 2, 3)
>>> a[0]
1
>>> a[0] = 10
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

A tuple is a set of values enclosed in "(" and ")". The major difference between a tuple and a list is that it is not possible to modify the tuple. This means it's impossible to do things like adding more elements to the tuple or modifying existing values.

10.3 What is an Associative Array?

Suppose you wish to represent a telephone directory in a Python program:

```
>>> d = {'Rahul':2150438, 'Roy':5128431, 'Vimal':7124196}
>>> d['Vimal']
7124196
>>> s = 'Roy'
>>> d[s]
5128431
>>>
```

A Python *associative array* (also called a *dictionary*) is created by enclosing key/value pairs in "{" and "}". In every key/value pair, the key and value should be separated by a colon. In the above case,:

```
'Rahul':2150438
```

is one key/value pair with key being the string 'Rahul' and value being the number 2150438. Given a key, it is easy to extract a value; you just have to index the associative array with that key.

10.4 Processing lists/tuples with *for* loops

Examining each element of a list/tuple in sequence and performing some action based on the value of that element is a common programming problem. For example, let's say you are given a list:

```
[10, 11, 14, 13, 17, 24, 78]
```

and asked to write some code to select the odd numbers (11, 13 and 17, in this case).::

```
>>> a = [10, 11, 14, 13, 17, 24, 78]
>>> n = len(a)
>>> i = 0
>>> while (i < n):
...     if((a[i] % 2) == 1):
...         print a[i]
...     i = i + 1
...
>>>
```

Here is a shorter (and clearer) version of the code using what is called a *for* loop:

```
>>> a = [10, 11, 14, 13, 17, 24, 78]
>>> for k in a:
...       if ((k % 2) == 1):
...           print k
...
>>>
```

The general form of a *for* loop is:

```
for variable_name in sequence:
    body
```

In the above case, the *variable_name* is *k* and the *sequence* is the list *a*; the body of the *for* loop is an *if* statement. This is the way the *for* loop works: the statements in the body get executed as many times as there are elements in the given sequence. In our example, the body gets executed seven times. Each time the loop executes, variable *k* will be assigned value of the next element in the sequence; so *k* will be initially 10, then it becomes 11 (in the next iteration) and so on till 78.

The *for* loop is not a complete replacement for the *while* loop; it is useful only in those situations where you have to process the individual elements of a sequence like a list or a tuple. In all other cases, a *while* loop is the natural choice.

CHAPTER

ELEVEN

PROGRAMMING WITH CLASSES AND OBJECTS

Besides writing correct code, a programmer has to make sure that he writes clear, readable code; code which others can read, understand and modify easily. One way to do this is by structuring the program as a collection of small functions. In this chapter, we examine another powerful technique for achieving the same goal.

Note: We are going to examine ideas which are a bit more complicated than what we have seen so far ... don't worry if you do not understand everything clearly.

11.1 Representing a student

As discussed in the previous chapter, you can represent a student in a Python program by a list:

```
>>> p = ["Tom", 15, 78]
>>> q = ["Jerry", 14, 81]
>>> p[0]
'Tom'
>>> q[0]
'Jerry'
>>> p[1]
15
>>> q[1]
14
>>> p[2]
78
>>> q[2]
81
>>>
```

We have two students, *Tom* and *Jerry* aged 15 and 14 respectively. They have scored marks 78 and 81 in the maths exam.

Let's write a Python function to find out the name of the student with higher mark:

```
>>> def better_student(a, b):
...       if(a[2] > b[2]):
...          return a[0]
...       else:
...          return b[0]
...
>>> better_student(p, q)
'Jerry'
>>>
```

The *better_student* function works perfectly; it compares the marks of the two students and returns name of the student with greater mark.

Imagine a situation where *better_student* is just one function in a big program composed of hundreds of functions. How does another programmer reading your code know that *a[2]* and *b[2]* represents the marks of the students without having to check out the definitions of *p* and *q* which might be at a location far removed from where you are defining *better_student*? Even when he sees the actual definition of *p* and *q*, how does he know that *p[2]* and *q[2]* are the marks (rather than *p[1]* and *q[1]*)? Surely, we need a better way to represent our student!

11.2 Using classes and objects

Look at the following Python program:

```
>>> class student:
...    pass
...
>>> a = student()
>>> a.age = 15
>>> a.mark = 78
>>> a.name = "Tom"
>>> b = student()
>>> b.age = 14
>>> b.mark = 81
>>> b.name = "Jerry"
>>> a.age
15
>>> b.mark
81
>>>
```

The first two lines constitute a *class declaration*. What does it do?

Any data item in a Python program belongs to one of several categories. For example, the number 12 belongs to the category *integer*, the sequence "hello" belongs to the category *string*, the number 1.23 belongs to the category *float*. The class declaration simply tells Python:

```
We have a new category - let's call it student.
```

The statement:

```
a = student()
```

tells Python to create a new student; in programming language terminology, we will call *a* an *object*.

The remaining lines:

```
a.age = 15
a.mark = 78
a.name = "Tom"
```

simply instruct Python to:

```
set the age, mark and name of the new student to
15, 78 and "Tom".
```

age, *mark* and *name* are referred to as *attributes* of the object.

We should examine the class declaration a bit more in detail. The first line of a class declaration is the keyword *class* followed by a name:

```
class name_of_class:
```

The body of a class declaration has to be indented (similar to functions, while loops and if statements). The simplest class is one which has an empty body; in such cases, we should write:

```
pass
```

in the place of the body. An empty line should be typed as the last line of the class declaration (similar to functions, while loops etc).

11.3 Rewriting *better_student*

Let's now rewrite *better_student* assuming that students are represented as objects of class *student*:

```
>>> def better_student(p, q):
...     if (p.mark > q.mark):
...          return p.name
...     else:
...          return q.name
...
>>> better_student(a, b)
'Jerry'
>>>
```

Note: The above code should be written after defining class student and creating two students *a* and *b*.

Which version of *better_student* is more readable? Obviously, the second one!

11.4 Improving the student class

What is wrong with the following Python code fragment?:

```
>>> m = student()
>>> m.age = 16
>>> m.name = 'Rahul'
>>> m.mark = 90
>>> n = student()
>>> n.age = 16
>>> n.name = 'Rohit'
>>> better_student(m, n)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in better_student
AttributeError: student instance has no attribute 'mark'
>>>
```

We haven't created an attribute *mark* for our student *Rohit*! The possibility for such errors can be reduced if there is some way to make sure that a student object is always created with the three attributes *age*, *name* and *mark* properly initialized. Here is a modified student class which takes care of this:

```
>>> class student:
...     def __init__(self, a, n, m):
...        self.age = a
...        self.name = n
...        self.mark = m
...
>>> m = student(16, "Rahul", 90)
```

```
>>> m.age
16
>>> m.name
'Rahul'
>>> m.mark
90
>>> n = student()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 4 arguments (1 given)
>>>
```

A class can have functions defined as part of its body. We are defining a function with a peculiar name:

```
__init__
```

(two underscore symbols, then the name *init* and then two more underscore symbols). The function takes four parameters:

```
self, a, n and m
```

When Python executes the statement:

```
m = student(16, "Rahul", 90)
```

it creates a student and immediately calls the function *__init__* passing 16 as the value of parameter *a*, "Rahul" as the value of *n* and 90 as the value of *m*. What does the parameter *self* refer to? Any function written within a class has to have a first parameter named *self* (Python does not require that this parameter be called specifically *self*, but it is a convention in the Python world to always use the name *self* as the name of the first parameter of a function defined within a class). When the function executes, *self* refers to the object as part of whose creation the *__init__* function was invoked. The job of the *__init__* function is to add the required attributes to the newly created object and initialize them properly. If you try to do something like this:

```
n = student()
```

Python will give you a syntax error - once you define a special function called __init__ as part of the body of a class, you will be able to create an object only by specifying values for all the parameters of the function (except *self*).

Note that you can define any kind of function within a class; only if your function is specifically called *__init__* will it be automatically called every time an object of that class is created. The special function *__init__* is called a *constructor* in programming language literature.

CHAPTER

TWELVE

EXCEPTION HANDLING

An error detected during the execution of a program is called an *exception*. This chapter briefly explains Python exception handling techniques.

Note: Exception handling is a complicated subject; a detailed exposition is beyond the scope of this book.

12.1 Syntax Errors

As a beginner, you will make lots of syntax errors while writing Python code:

```
>>> def sqr(x)
  File "<stdin>", line 1
    def sqr(x)
           ^
SyntaxError: invalid syntax
>>>
```

In the above case, the problem is with a missing colon after the first line of the function definition. Such errors are usually not very difficult to identify and correct.

12.2 Exceptions

It is more difficult to troubleshoot errors which occur during program execution. Let's look at a simple example.

The slope of a line segment with co-ordinates (x1, y1) and (x2, y2) is given by the equation:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Here is a Python function for computing the slope:

```
>>> def slope(x1, y1, x2, y2):
...     m = (y2 - y1) / (x2 - x1)
...     print 'slope computed'
...     return m
...
>>> slope(1.0, 1.0, 4.0, 4.0)
slope computed
1.0
>>>
```

What happens if you try to find the slope of a vertical line with co-ordinates (1.0, 1.0) and (1.0, 4.0)?:

```
>>> slope(1.0, 1.0, 1.0, 4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in slope
ZeroDivisionError: float division
>>>
```

We are getting an error; the last line of the message conveys the exact nature of this error - it is a zero division error. You will get the same error if you try doing:

```
1.0/0
```

at the Python prompt.

The moment Python discovers that the division is going to result in an error, it stops any further execution of the code and immediately prints an error message. The problem with this behaviour is that the programmer might wish to have some control over *how* his code should respond to the error. Here is a modified slope function:

```
>>> def slope(x1, y1, x2, y2):
...     try:
...         m = (y2 - y1) / (x2 - x1)
...         print 'computed slope'
...     except ZeroDivisionError:
...         print 'unable to compute slope'
...         m = 'undefined'
...     return m
...
>>> slope(1.0, 1.0, 4.0, 4.0)
computed slope
1.0
>>> slope(1.0, 1.0, 1.0, 4.0)
```

```
unable to compute slope
undefined
>>>
```

You can see two new keywords here: *try* and *except*. Python first attempts to execute statements between the *try* and the *except* (note that these statements are indented). If there is no zero division error, the *except block* (the indented set of statements after the line "except ZeroDivisionError:") is skipped and the remaining part of the function (which is only a single *return* statement) gets executed. If there is an error, the remaining statements in the *try block* (statements between *try* and *except* constitute the *try block*) are skipped and control gets transferred to statements in the *except block*. Once all the statements in the *except block* are executed, the remaining part of the function gets executed.

An informal description of this behaviour might be:

```
Try the following statements. Stop if there is an error  and skip to the
matching *except block*. If there is no error, complete the statements
(between *try* and *except*) and skip the *except block*.
```

What have we achieved here? We now have a choice to decide *what* should be done in case of an error. This is a basic requirement when writing real life code.

12.3 Exercises

1. Create a Python list and index it beyond its limit. Do you get an exception? What is its name?

CHAPTER

THIRTEEN

HANDLING DATA FILES

Operating systems use files for permanent data storage (usually on a hard disk, CD/DVD or thumb drive). Python programs can read data stored in files; they can also write data to files. In this chapter, we will examine how this is done.

13.1 Writing a message to a file

Here is a simple Python program which writes a message to a file:

```
>>> f = open('abc', 'w')
>>> f.write('good morning!')
>>> f.close()
>>>
```

The *open* function takes two parameters - the first one is the name of a file (in this case, *abc*) and the second one is a *mode* (in this case, 'w' which means we wish to open the file for writing). *Open* returns an object which is stored in *f*; if you wish to write a message (say 'good morning!') to the file, simply execute:

```
f.write('good morning!')
```

Note: You can also store the string in a variable and pass the variable as parameter to f.write

If you are familiar with the GNU/Linux operating system, you will see a file called *abc* when you browse the file system!

Once a message is stored in a file, it can be easily retrieved any time:

```
>>> g = open('abc', 'r')
>>> s = g.readline()
>>> print s
'good morning!'
>>>
```

We are now opening the file for *reading* (note the 'r' as second parameter to *open*). The function:

```
g.readline()
```

reads one line from the file; it is then stored in *s*.

13.2 Writing and reading multiple lines

What does the following Python program do?:

```
>>> f = open('data', 'w')
>>> f.write('hello\n')
>>> f.write('world\n')
>>> f.write('python\n')
>>> f.close()
```

It opens a file 'data' for writing and writes three strings. Note the 'n' at the end of the strings - it stands for the *newline* character (the character corresponding to the *Enter* key). We want the three strings to be stored as three independent lines in the file.

We can read these lines back:

```
>>> g = open('data', 'r')
>>> a = g.readlines()
>>> print a
['hello\n', 'world\n', 'python\n']
>>>
```

The function:

```
g.readlines()
```

reads all the lines in the file and returns a list containing these lines.

13.3 Writing and reading numerical data

Let's try to store a number 10 into a file:

```
>>> f = open('dat2', 'w')
>>> i = 10
>>> f.write(i)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
```

```
TypeError: argument 1 must be string or read-only character buffer, not int
>>>
```

This does not work; the *write* function expects a string as its parameter and will not work with any other type. A solution is to convert the integer 10 to a string and then write it to the file:

```
>>> f.write(str(i))
>>> f.close()
```

The *str* function returns a string representation of the value in *i* and it is then written to the file.

You have to be careful when you read back the data:

```
>>> g = open('dat2', 'r')
>>> s = g.readline()
>>> i = int(s)
>>>
```

The *readline* function returns the string representation of the integer stored in the file. It has to be converted back to its original form by calling the function *int*.

CHAPTER

FOURTEEN

INTRODUCTION TO THE LATEX DOCUMENT PREPARATION SYSTEM

What will you do if your teacher asks you to prepare a project report using the PC? You will open Microsoft Office, type the document, and save it as a *doc* file. You will spend a good amount of time changing fonts, aligning the lines and in general trying to make the document look "beautiful".

There are many problems with this approach to document preparation especially as you start writing lengthy (and technical) documents like scientific reports or books. It is a bit difficult at this point to explain what these problems are; we shall get into it later. The important thing is to realize that using a word processor like MS Office (or OpenOffice, on GNU/Linux) is not the only way to prepare documents. A far superior method is available; this involves using a program called LaTeX.

Note: We will make heavy use of the operating system terminal/command line as well as a text editor program like *gedit* in the remaining part of this book. So it is a good idea to review the material in Chapter 1 as well as Appendix 1 before proceeding further.

14.1 A Simple LaTeX document

Use your favourite text editor (say *gedit*) and create a text file containing the following lines:

```
\documentclass{article}
\title{Srinivasa Ramanujan}

\begin{document}

\maketitle

Srinivasa Ramanujan was an Indian mathematician and
self taught genius who, with almost no formal training
in pure mathematics, made substantial contributions to
mathematical analysis, number theory,
```

```
infinite series and continued fractions.

\end{document}
```

Save the file as *ramanuja.tex*; this is your report on Srinivasa Ramanuja, the famous Indian mathematician, which you are supposed to submit to your maths teacher tomorrow!

But wait, what are all those mysterious \documentclass, \title etc doing in the document?

The idea is that you can use a powerful program called LaTeX to convert this file into a very good looking document which can be printed on paper. How do we do this?

Open a terminal and type the following command:

```
pdflatex ramanuja.tex
```

"pdflatex" generates a new file "ramanuja.pdf" using the data contained in "ramanuja.tex"; you will see a lot of messages getting printed on the screen during this process - don't worry about it as long as you get back the *command prompt* (in case there is some error, you will not get the command prompt - we will see how to handle this towards the end of this chapter).

A file with name ending in .pdf can be given directly to a printer or it can be viewed on the screen (on GNU/Linux) using one of two programs: either *xpdf* or *evince*. You simply have to give the command:

```
xpdf ramanuja.pdf
```

or:

```
evince ramanuja.pdf
```

at the operating system command line.

Here is what you will see when you print (or view) the PDF document:

Srinivasa Ramanujan

January 24, 2010

Srinivasa Ramanujan was an Indian mathematician and self taught genius who, with almost no formal training in pure mathematics, made substantial contributions to mathematical analysis, number theory, infinite series and continued fractions.

Sometimes, your GNU/Linux installation may not have the command *pdflatex* installed. In that case, there is another way to generate printable output.

First, we run the command *latex*:

```
latex ramanuja.tex
```

This will generate a new file called "ramanuja.dvi". You have to convert a .dvi file to either PDF or Postscript by using the commands *dvipdf* or *dvips*:

```
dvips ramanuja.dvi
```

will generate a new Postscript file called "ramanuja.ps" which can be directly printed or viewed on the screen using *xpdf* or *evince*.

If you wish to generate PDF output, you can try:

```
dvipdf ramanuja.dvi
```

14.2 What does LaTeX really do?

If you look at the PDF document produced by LaTeX, you will see that it does not contain any of the lines \documentclass, \title etc. Instead, what you see is neatly formatted output. The title has a bigger font size and is centralized. The left and right ends of all the lines are perfectly aligned; the word "continued" has been split to make the alignment perfect. Usually, when we prepare documents using MS-Word or OpenOffice, we spend a lot of time doing all these things manually - the problem with the approach is that as the document gets larger in size, we will be spending more and more time on this rather than concentrating on our writing. It's here that LaTeX comes to our rescue. All the details involved in getting beautiful printed output is handled by LaTeX - we simply have to focus on our writing! This does not mean that we can simply type an article and expect LaTeX to format it properly. For example, how does LaTeX know what the title of the article is?

Simple. When you write a line like:

```
\title{Srinivasa Ramanujan}
```

you are telling LaTeX that "Srinivasa Ramanujan" is the title of the article. This does not result in the title being actually displayed - LaTeX simply notes down the fact that the title is "Srinivasa Ramanujan". It is displayed when you type:

```
\maketitle
```

Because it is the title, LaTeX gives it a bigger font size and displays it at the center.

Now, how does LaTeX know what font size is to be used in the body? If you are preparing a document which is to be used as a presentation and projected onto a screen (say using an LCD projector), the fonts have to be much larger. This is where the "documentclass" becomes relevant. If you look at the kind of documents around you, you will see articles, books, letters, slides etc. Each "kind" of document has its own logical structure. A letter has a "From Address", "To Address", "Signature", "Salutation", "Subject", "Place", "Date" etc while an article does not have most of these. A book will have chapters, sections and sub-sections while an article won't have chapters in it. Before you start writing the content of a LaTeX document, you must tell LaTeX what kind of document you are planning to write. That is what:

```
\documentclass{article}
```

does; you are telling LaTeX that what you are going to prepare is an article and NOT a slide or a book or a letter. Once you give this information to LaTeX, LaTeX will decide *how* your document is to be formatted.

Note: You can specify options to the \documentclass command; say you wish the font to be 12 point in size and the display be in two columns, you can write: \documentclass[twocolumn,12pt]{article}

Once you specify the document class and title, you can write the actual content of the document - this has to be written between the lines:

```
\begin{document}
```

and:

```
\end{document}
```

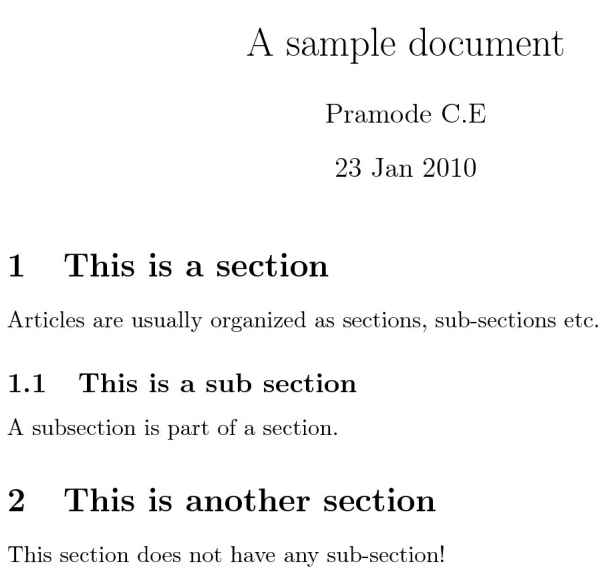
14.3 Sections and sub-sections

Create another text file, *sample1.tex*, with the following lines in it:

```
\documentclass {article}
\title{A sample document}
\author{Pramode C.E}
\date{23 Jan 2010}

\begin{document}

\maketitle

\section{This is a section}
```

```
Articles are usually organized as sections, sub-sections etc.

\subsection{This is a sub section}

A subsection is part of a section.

\section{This is another section}

This section does not have any sub-section!

\end{document}
```

This is how the generated pdf/postscript file looks:



A sample document

Pramode C.E

23 Jan 2010

1 This is a section

Articles are usually organized as sections, sub-sections etc.

1.1 This is a sub section

A subsection is part of a section.

2 This is another section

This section does not have any sub-section!

The \title, \author and \date commands set a title, author and date for the document (current date is automatically used if date is not specified) - these are displayed only when \maketitle is invoked.

A new section is specified by writing:

```
\section{name of section}
```

LaTeX automatically numbers the section and chooses an appropriate font size. Likewise, a sub-section is created by:

```
\subsection{name of sub-section}
```

Note that LaTeX again saves us the effort involved in section numbering and font size selection. LaTeX is also capable of automatically generating a table of contents for us using the section names specified!

14.4 Lists, Quotes and Quotations

Enter the following text into a file, say *sample2.tex*:

```
\documentclass {article}

\begin{document}

Here is how you typeset a short quotation:
\begin{quote}
This is an example of a short quotation. It consists of a single
para of text. See how it is formatted.
\end{quote}

And, here is an example of a longer quotation:
\begin{quotation}
This is an example of a longer quotation. It consists of more
than one paragraph of text.

This is the second para of the quotation.
\end{quotation}

Here is an example of an unnumbered list:
\begin{itemize}
 \item Apple
 \item Orange
 \item Banana
\end{itemize}

And, an example of a numbered list:
\begin{enumerate}
 \item Newton
 \item Pascal
 \item Euler
\end{enumerate}

\end{document}
```

Here is the PDF output produced by LaTeX:

Here is how you typeset a short quotation:

> This is an example of a short quotation. It consists of a single para of text. See how it is formatted.

And, here is an example of a longer quotation:

> This is an example of a longer quotation. It consists of more than one paragraph of text.
>
> This is the second para of the quotation.

Here is an example of an unnumbered list:

- Apple
- Orange
- Banana

And, an example of a numbered list:

1. Newton
2. Pascal
3. Euler

A short quotation of only one paragraph can be formatted by enclosing it within:

```
\begin{quote}
```

and:

```
\end{quote}
```

The \begin command is said to introduce a LaTeX *environment*, the name of the *environment* is what is given in curly brackets. Note that LaTeX offsets the text within the *quote* a little bit to the right.

The *quotation* environment should be used for displaying longer (multiple paragraphs) quotations.

The *itemize* environment is used for displaying unnumbered lists. Each item in the list should be written like this:

```
\item name of item
```

If you want numbered list, you can use the *enumerate* environment.

14.5 Handling errors

Try this experiment: in *sample2.tex*, change:

```
\end{enumerate}
```

to:

```
\end{itemize}
```

and run *pdflatex* or *latex* once again. Instead of getting back the command prompt, you will see something like this:

```
! LaTeX Error: \begin{enumerate} on input line 27 ended by \end{itemize}.

See the LaTeX manual or LaTeX Companion for explanation.
Type  H <return>  for immediate help.
...

l.31 \end{itemize}

?
```

The error occurs because a \begin{enumerate} should end with \end{enumerate}. You should type **q** to get back the command prompt and rectify the error.

CHAPTER

FIFTEEN

TYPESETTING MATHEMATICS

A big advantage of LaTeX is that it can be used to prepare documents containing complex mathematical formulae. In this chapter, we examine how to typeset mathematics with LaTeX.

15.1 The three environments - math, displaymath and equation

Let us typeset an equation in three different ways. Here is the LaTeX document:

```
\documentclass {article}
\begin{document}


Here is a formula: $E = mc^{2}$

Same formula, but this is typeset differently: \[ E = mc^{2} \]

Same formula again, different typesetting:
 \begin{equation}
 E = mc^{2}
 \end{equation}

Repeat:
 \begin{equation}
  E = mc^{2}
 \end{equation}

\end{document}
```

This is the output which LaTeX generates:

Here is a formula: $E = mc^2$
Same formula, but this is typeset differently:

$$E = mc^2$$

Same formula again, different typesetting:

$$E = mc^2 \tag{1}$$

Repeat:

$$E = mc^2 \tag{2}$$

When an equation is written between two "$" (dollar) signs, it is said to be in *math* mode. In the above document:

```
$E = mc^{2}$
```

is in math mode. The equation is displayed in the same line (it is called an in-text formula). When the same equation is written between a "\[" and "\]", it is said to be in *displaymath* mode; it is displayed on a line of its own.

When the equation is written between a "\begin{equation}" and "\end{equation}", LaTeX gives it an equation number - the first equation is given the number 1 and subsequent equations are numbered 2, 3 and so on.

Now, how do you really write an equation? Mathematical equations are full of symbols which we rarely use when writing normal text - LaTeX has a syntax of its own for representing such symbols and symbol combinations. The equation:

$$E = mc^2$$

itself is a good example. LaTeX displays the number 2 as a "superscript" when the "^" operator is used.

15.2 A few examples

Given below are a few samples of the kind of mathematical expressions you can create using LaTeX (the expressions have to be in one of math, displaymath or equation modes - that is omitted in the examples).

LaTex Command:

```
x_{i}
```

result:

$$x_i$$

LaTeX command:

```
p = \frac{a+b}{a-b}
```

result:

$$p = \frac{a+b}{a-b}$$

LaTeX command:

```
\sqrt{a+b}
```

result:

$$\sqrt{a+b}$$

LaTeX command:

```
\sqrt[n]{2}
```

result:

$$\sqrt[n]{2}$$

LaTeX command:

```
\alpha \beta \pi
```

result:

$$\alpha\beta\pi$$

LaTeX command:

```
\leq
```

result:

$$\leq$$

LaTeX command:

```
\not\leq
```

result:

$$\not\leq$$

LaTeX command:

```
\sum_{i=1}^{n} x_{i} = \int_{0}^{1} f
```

result:

$$\sum_{i=1}^{n} x_i = \int_0^1 f$$

LaTeX command:

```
\lim_{n \rightarrow \infty} x = 0
```

result:

$$\lim_{n \to \infty} x = 0$$

LaTeX command:

```
\overline{a+b}
```

result:

$$\overline{a+b}$$

LaTeX command:

```
\underline{a+b}
```

result:

$$\underline{a+b}$$

LaTeX command:

```
\overbrace{a+b+c+d}
```

result:

$$\overbrace{a+b+c+d}$$

CHAPTER

SIXTEEN

PICTURES AND COLOURS IN LATEX

It is possible to include pictures in your LaTeX document. It is also possible to draw simple figures and change colours using LaTeX commands itself. This chapter explains how this is done.

Note: Using an additional package called "pstricks", it is possible to generate complex figures (like plots of equations) in LaTeX. A simple "pstricks" example is given in the last section of this chapter.

16.1 The LaTeX "picture" environment

Create a file called *pic1.tex* containing the following lines:

```
\documentclass {article}
\begin{document}

\section{This is a section}

\begin{picture}(100,200)

\put (0,0) {A}
\put (100,0) {B}
\put (100, 200) {C}
\put (0, 200) {D}
\end{picture}

\end{document}
```

Here is the PDF/Postscript output produced by LaTeX:

The LaTeX command:

```
\begin{picture}(100, 200)
```

creates a "picture environment" of width 100 units and height 200 units (imagine the picture environment producing a 100x200 box) - a unit has a default value of about 0.35mm. The command:

```
\put (0,0) {A}
```

writes the symbol "A" at location (0,0) of the box; location (0,0) is the (x, y) co-ordinate of the lower left corner of the box (it is the origin). The +ve X axis is towards the right and the upward direction represents +ve Y axis. The command:

```
\put (100,200) {C}
```

places the symbol C at the top right corner. The commands:

```
\put (100, 0) {B}
\put (0, 200) {D}
```

places the symbols B and D at the bottom right and top left corners of the box.

16.2 Drawing Lines

Drawing lines in LaTeX is tricky business - due to certain limitations, it is not possible to draw lines with any random slope. The natural way for a line drawing command to work would be to accept the co-ordinates of the end points and draw a line connecting the points. But that is not the way it is done in LaTeX. To simplify things, we consider three separate cases: drawing a horizontal line, drawing a vertical line and drawing lines with many other slopes. For all three cases, we assume a box of size 100x200 with origin (0, 0) at the bottom left corner (the same box as in the previous section).

16.2.1 Horizontal lines

Let's say we wish to draw a horizontal line of length 40 units from point (30, 90). Here is how it is done:

```
\put(30,90){\line(1,0){40}}
```

We can identify two separate parts in this command; the first part:

```
\put(30, 90)
```

says: *start the line at point (30, 90)*. The next part is:

```
\line(1,0){40}
```

This is how it works: LaTeX draws a line of length 40 units and passing through the points:

```
(30, 90) and (30 + 1, 90 + 0)
```

and ending at (70, 90).

What if we write:

```
\put(30, 90){ \line(-1, 0) {30} }
```

We get a line passing through (30, 90) and (30 - 1, 90 + 0) of length 30 units, ending at (0, 90).

So, the general command for drawing a horizontal line is:

```
\put(x,y) { \line(A, 0){len} }
```

Where (x,y) is the starting co-ordinate of the line, A is +1 or -1 (depending on whether you want to draw to the right of (x, y) or left of (x, y) and *len* is the length of the line.

16.2.2 Vertical lines

It should be easy to guess how vertical lines are drawn! Here is an example:

```
\put(30,90) {\line(0,1) {40}}
```

This draws a vertical line connecting (30, 90) and (30, 91) of length 40 units ending at (30, 130). The general command is:

```
\put(x,y) { \line(0, A){len} }
```

Where (x, y) is the starting co-ordinate of the line, A is +1 or -1 depending on whether the line goes up or down and *len* is the length of the line.

16.2.3 Slanted lines (neither vertical nor horizontal)

What kind of line does LaTeX draw if we write:

```
\put(30,90) {\line(1, -1) {40}}
```

LaTeX draws a line going through (30, 90) and (31, 89) (ie, sloping down to the right); what is the length of this line? That is not specified here - the number 40 written in curly brackets simply indicates the fact that as the line moves down to the right, it will end at that point where the X co-ordinate is 30 + 40.

What if we write:

```
\put(40,90) {\line(-1, 1) {30}}
```

We get a line starting at (40, 90), going through (39, 91) (going up to the left) and ending at a point whose X co-ordinate is 40 - 30, ie 10.

The general form of the command for a slanted line is:

```
\put(x,y) {\line(p, q) {len}}
```

LaTeX has certain restrictions on the values of p and q - this places a limit on the slopes of the lines which we can draw. Both p and q should be integers between -6 and +6, inclusive. Also, they should have no common divisor bigger than 1. That is, p/q should be a fraction in its simplest form - you can't have something like p = 2 and q = 4; you should write p = 1 and q = 2. The following are all illegal values for p and q:

```
(1.2, 3) --- no decimal permitted
(3, 6) --- common divisor 3 bigger than 1
(1, 7) --- one value bigger than +6
```

The smallest slanted line which LaTeX can draw is a line of length 10 points (about 3.5mm). LaTeX will draw nothing if you try to draw slanted lines less than this length.

16.3 Circles, ovals and bezier curves

Note: We are assuming that the picture environment has width 100 units and height 200 units in all the examples below.

Here is how you can draw a circle:

```
\put(20,30) {circle{20}}
```

This draws a circle at center (20,30) with diameter 20 points.

The command:

```
\put(20,30) {\circle*{20}}
```

draws a disc (a filled circle) at (20,30) with a diameter of 20 points.

LaTeX knows to draw discs/circles with only a certain fixed number of diameters - it will choose the one whose diameter is closest to what you have specified. Also, the set of possible diameters has an upper limit.

The command:

```
\put(30,30) {\oval(20,10)}
```

will draw an oval (a rectangle with rounded corners) of width 20 units and height 10 units.

The qbezier command takes three points as arguments and draws a quadratic *bezier curve* connecting them:

```
\qbezier (0,0) (50,100) (100, 0)
```

16.4 Using the "graphics" package

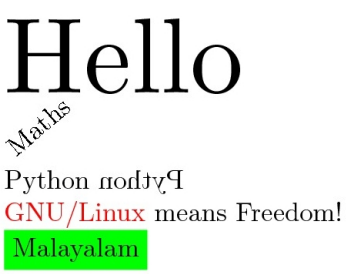
Using a package called "graphics", we can include images in our LaTeX document, perform manipulations like rotation/scaling, add colour etc.

16.4.1 Scaling, Rotation, Colour change

Create a file called *pic2.tex* with the following lines in it:

```
\documentclass {article}
\usepackage{graphics}
\usepackage{color}


\begin{document}

\scalebox{4}{Hello}

\rotatebox{40} {Maths}

Python \reflectbox{Python}

\textcolor{red}{GNU/Linux} means Freedom!

\colorbox{green}{Malayalam}

\end{document}
```

Here is the output produced by LaTeX:



The two lines at the beginning:

```
\usepackage{graphics}
\usepackage{color}
```

are essential for commands like \scalebox, \textcolor etc to work. You can think of them as being similar to the *import* statement in Python - they make available additional functionality.

You can enlarge text by a constant scale factor using \scalebox. For example:

```
\scalebox{4}{Hello}
```

displays “Hello” enlarged by a factor of 4.

You can rotate text using \rotatebox. For example:

```
\rotatebox{40}{Maths}
```

displays the text “Maths” 40 degree rotated.

The \reflectbox command generates a mirror image. For example:

```
\reflectbox{Python}
```

displays the mirror image of the string “Python”.

Color of text can be changed using \textcolor. For example:

```
\textcolor{red}{GNU/Linux}
```

displays the string “GNU/Linux” in red colour.

A string can be displayed in a coloured box using \colorbox.

16.4.2 Displaying images

Sometimes, you may have to include an external image file (say a JPG or PNG file) in your LaTeX document. There are two ways to do it.

Suppose you are using the “pdflatex” command and want to display a photo of Ramanujan somewhere in your document. Just write:

```
\includegraphics{ramanujan.jpg}
```

Note that “pdflatex” supports only JPG and PNG formats. The file “ramanujan.jpg” should exist in the folder(directory) from where you are issuing the “pdflatex” command.

If you are using the “latex” command, you will have to first convert your JPG/PNG image to what is called an *encapsulated postscript* file. This can be done very easily on GNU/Linux systems by using a command called “convert”. At the command prompt, you should type:

```
convert ramanujan.jpg ramanujan.eps
```

Once this is done, you can use:

```
\includegraphics{ramanujan.eps}
```

to include the image in your document.

16.5 Using "pstricks" for advanced picture drawing

We have seen that the facilities available by default in LaTeX for drawing lines, circles etc are not very sophisticated. A package called "pstricks" can be used for generating complex figures without too much trouble. Let's try to plot a curve using pstricks:

```
\documentclass {article}
\usepackage{pstricks}
\usepackage{pst-plot}

\begin{document}


\begin{pspicture}(-3, -2)(3, 2)
\psaxes(0,0)(-3,-2)(3,2)
\psplot[plotstyle=curve] {-1.5} {1.5} {x 3 exp x sub}
\end{pspicture}

\end{document}
```

Two packages have to be included: *pstricks* and *pst-plot*. Also, "pdflatex" will not work with the above file - you have to use the "latex" command itself.

The line:

```
\begin{pspicture} (-3, -2) (3, 2)
```

starts a "pspicture" environment - think of it as a request to LaTeX to leave enough space for a rectangle whose bottom left corner has co-ordinate (-3, -2) and top right corner has co-ordinate (3, 2).

The next command:

```
\psaxes(0,0) (-3, -2) (3, 2)
```

draws the co-ordinate axes. The X axis and Y axis meet at the point (0, 0) and they have to be visualized as being enclosed in an imaginary rectangular box with bottom left corner at (-3, -2) and top right corner at (3, 2).

We are plotting the curve:

```
y = (x * x * x) - x
```

This equation has to be specified in a peculiar form called *postfix*. An arithmetic expression written in the usual way:

```
2 * 3 + 4
```

is called an *infix expression*. Here is another way to write the same expression:

```
2 3 * 4 +
```

This is called a *postfix expression*. The logic is simple. Read the expression from left to right. When you encounter an operator, simply apply the operator to the two operands to the left. In the above case, the moment we see the * operator (multiplication), we can rewrite the expression as:

```
6 4 +
```

Now, when we encounter the + operator, we can rewrite the expression as:

```
10
```

Let's examine this line in our LaTeX document:

```
\psplot[plotstyle=curve] {-1.5} {1.5} {x 3 exp x sub}
```

We are asking LaTeX to plot a curve; the curve is given by the equation:

```
y = x 3 exp x sub
```

(note: you need to specify only the right hand side of the equation).

This equation is written in postfix form; *exp* is the exponentiation operator and *sub* is the subtraction operator. So we can read this as:

```
y = (x raised to 3) minus x
```

The numbers -1.5 and +1.5 in brackets refers to the range of possible values for x.

CHAPTER

SEVENTEEN

APPENDIX 1 - WRITING STAND-ALONE PYTHON PROGRAMS

Till now, we were using the interactive Python prompt to write short programs. The disadvantage with this method is obvious - all the code that you write is lost when you exit from Python. In this chapter, we shall see how to store Python programs permanently in text files.

17.1 Using an editor in GNU/Linux

An *editor* is a program which helps to create text files which can be saved permanently in the computer's hard disk. The GNU/Linux operating system comes with many editors; two of the most frequently used are *vi* and *emacs*. Both of them are a bit difficult to use for absolute beginners. A better choice might be a program called *gedit*. You can run *gedit* by simply typing **gedit** at the operating system command line (refer chapter 1 for a discussion about the command line).

gedit is easy to use. You can type whatever you want; once you are finished with typing, it's time to save the data you have typed. For this, choose the *Save* option from the *File* menu; *gedit* will now display a new window where you will be asked to give a name to the file in which you wish to save the typed data. Choose the *Quit* option from the *File* menu to exit from *gedit*.

17.2 Writing a Python program using *gedit*

Run *gedit* and save the following line into a file called *hello.py*:

```
print "Hello"
```

Quit *gedit* and type the following at the Operating System commandline:

```
python hello.py
```

You will see the message "Hello" getting printed on the screen.

Create another file using *gedit* and store the following code in it (you may name the file abc.py - the .py at the end tells us that we are dealing with a file containing Python code):

```
def sqr(x):
   return x*x
m = sqr(3)
print m
```

You can run the code by typing:

```
python abc.py
```

at the operating system commandline.

Note: You should be more careful when writing Python code in this way. When you are typing code at the Python prompt, you get immediate feedback in case of syntax errors. That's not the case when you store and execute Python code from a file.

17.3 Performing keyboard input

Using *gedit*, store this program in a file called *test.py*:

```
a = input()
b = input()
print a * b
```

Run the code by typing, at the operating system commandline:

```
python test.py
```

The *input* function is waiting for you to enter some data from the keyboard; type the number 10 and hit the *Enter* key. The number will be read and stored in the variable *a*. Now type one more number (say 20) and hit *Enter*; it will be stored in *b*. The program will print 200 as the output.

CHAPTER

EIGHTEEN

LICENSE

This book is published under the GNU FDL

CHAPTER

NINETEEN

HOW TO BUILD THE BOOK FROM SOURCE

This book is written using an excellent tool called Sphinx. Sphinx allows you to write in simple restructured text format - the conversion to HTML (for display in a browser) as well as LaTeX (for printing) is done by Sphinx. You are reading the Sphinx generated book right now! Source files are available from GitHub.

A little bit about my book-writing work flow. I maintain a git repo on my home machine from where I occasionally push to GitHub. I have another git repository on my Linode slice which hosts the book - occasional pulls from github combined with a build which directly delivers HTML files to the web server's (I use Nginx - do try it out if you find Apache to be a memory hog) static files folder is all that is needed to "publish" the book!