

Abstract

This book will help you to learn the Python programming language, whether you are new to computers or an experienced programmer. let's begin!

Table of Contents

Abstract	1
Preface.....	3
Introduction	3
Who This Book is For	4
How I Got Into Using Python	4
Status of the book	4
Official Website	4
License Terms	5
Feedback	5
Acknowledgements	5
Something To Think About	5
Introduction.....	5
Introduction	5
Features of Python	6
Why not Perl?	7
What Programmers Say	8
Installing Python.....	8
For Windows users	8
For Linux/BSD users	9
For Mac users	9
Summary	10
First Steps.....	10
Introduction	10
Using the interpreter prompt	10
Choosing an Editor	11
Using Source Files	12
Comments	13
Executable Python programs	13
Getting Help	15
Summary	15
Basics.....	15
Introduction	15
Literal Constants	15
Numbers	16
Strings	16
Variables	16
Identifier naming	16
Data Types	17
Objects	17
Example of using Variables and Literal Constants	17
Logical and Physical Lines	18
Indentation	19
Summary	20

Operators and Expressions.....	20
Introduction	20
Operators	21
Operator Precedence	23
Order of Evaluation	25
Associativity	25
Expressions	25
Summary	26
Control Flow.....	26
Introduction	26
The if statement	26
The while statement	28
The for loop	29
The break statement	30
The continue statement	31
Summary	32
Functions.....	32
Introduction	32
Function parameters	33
Local Variables	34
The global statement	34
Default argument values	35
Keyword arguments	36
The return statement	37
DocStrings	37
Summary	38
Modules.....	39
Introduction	39
Byte-compiled .pyc files	41
The from...import statement	41
A module's <code>__name__</code>	41
Making your own modules	42
The <code>dir()</code> function	43
Summary	44
Data Structures.....	44
Introduction	44
List	44
Tuple	46
Dictionary	48
Sequences	49
References	51
Summary	52
Problem Solving.....	53
Introduction	53
The problem	53
The Solution	54
Phases	60
Summary	60
Object oriented programming.....	61
Introduction	61
The self	62
Classes	62

Object Methods	63
The <code>__init__</code> method	63
Class and Object Variables	64
Inheritance	66
Summary	68
Input Output.....	69
Introduction	69
Files	69
Pickle	70
More about Strings	72
Summary	73
Exceptions.....	74
Introduction	74
Errors	74
Handling Exceptions	75
Raising Exceptions	75
Try...Finally	76
Summary	77
Standard Library.....	78
Introduction	78
The <code>os</code> and <code>sys</code> modules	78
Summary	80
More Python.....	81
Introduction	81
Special Methods	81
Single statement blocks	82
List Comprehension	82
Receiving tuples and lists in functions	83
Lambda forms	83
The <code>exec</code> and <code>eval</code> statements	84
The <code>assert</code> statement	84
The <code>repr</code> function	84
Summary	84

Preface

Introduction

A programming language is a way of writing *programs*. Programs are a set of steps to make your computer do something useful. The program can be run again and again. Programs can do useful things such as automating your chores that can be done on a computer and do calculations quickly, and so on. There are plenty of reasons why learning how to program a computer can be useful for you.

Python is one such programming language. What makes it special is that it is both simple and powerful, which is a surprisingly rare combination. This makes it appealing for both beginners as well as experts, and more importantly, makes it fun to program with. This book aims to help you learn this wonderful language, and show how to get things done quickly and painlessly.

Who This Book is For

This book serves as a guide or tutorial to the Python programming language. It is mainly targeted at newbies (those who are new to computers). It is also useful for experienced programmers who are new to Python.

The aim is: If all you know about computers is how to open and save text files, then you should be able to learn Python from this book. If you have previous programming experience, then this book should be useful to get you up to speed on Python.

For experienced programmers, I have highlighted differences between Python and few other programming languages. A little warning though, Python is soon going to become your favorite programming language!

How I Got Into Using Python

I first started with Python in 2002 when I had to write an installer for my college project called Diamond. I had two choices - the Python and Perl bindings for the Qt library (we will get into what bindings are later in the book). A few web searches led me to an article by Eric S Raymond called [Why Python?](#) about how Python had become his favorite programming language. Later, I found out that the PyQt bindings were in a stable state as opposed to Perl-Qt. So, I decided that I had to learn Python and PyQt bindings to create the installer .

I started searching for a good book on Python - I couldn't find any. I did find some O'Reilly books but they were either too expensive or were more like a reference manual than a guide. So, I settled for the documentation that came with Python. The official tutorial gave a good idea about Python but was brief and small in its coverage. I was able to manage since I had previous programming experience, but I felt it was unsuitable for beginners.

About six months after my first brush with Python, I installed the latest Red Hat 9.0 Linux and I was fascinated by the improvements in the interface of KDE, especially KWord. I got excited about it and got the idea of writing something on Python. The plan was to write a few pages but soon I realized I had written close to 30 pages. That's when I got serious about writing a book. After several rewrites, it has reached a stage where I would say it has become a useful guide to learning the Python language. I consider this book to be my contribution and tribute to the open source community.

This book started out as my personal notes on Python and I still think of it in the same way, although I've taken a lot of effort to make it more palatable to others :-)

In the true spirit of open source, I have received lots of constructive suggestions, criticisms and feedback from enthusiastic readers which has helped me improve this book immensely.

Status of the book

This book is a *work-in-progress*. Many chapters are constantly being improved. However, this book has matured a lot. You should be able to learn Python easily from this book. Please do tell me if you find any part of this book to be incorrect or incomprehensible.

Ideas for more content for this book include topics such as debugging using [pdb](#).

Official Website

The official website of the book is <http://www.byteofpython.info>. From the website, you can read the whole book online, you can download the latest versions of the book, and also read the latest news about the book.

License Terms

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Basically, you are free to copy, distribute, and display the book, as long as you give credit to me. You are free to modify and build upon this work, provided that you clearly mark all changes and release the modified work under the same license as this book.

Feedback

Constructive suggestions, criticisms or even praise is welcome. My email address is *swaroop -at- swaroopch.info*.

Acknowledgements

Thanks to all the readers who have sent in their feedback and suggestions. This book would have never been good without you folks!

Something To Think About

- "There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies." -- C. A. R. Hoare
- "Success in life is a matter not so much of talent and opportunity as of concentration and perseverance." -- C. W. Wendte

Introduction

Introduction

Python is one of those rare languages which can claim to be both *simple* and *powerful*. You will find it easy to concentrate on the solution to your problem rather than having to concentrate on the programming language.

The official introduction to Python is

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

We will discuss most of these features in the next section.

Note

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Features of Python

Simple

Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English. This pseudo-code nature of Python is one of its greatest strengths.

Easy to Learn

As we will see, Python is extremely easy to get started with. It has an extraordinarily simple syntax.

Free and Open Source

Python is an example of a FOSS (Free and Open Source Software). In simple terms, you can freely distribute copies of this software, read its source code, make changes to the source code, and use pieces of it in new free programs.

FOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and constantly improved by a community which just wants to see a better Python.

High-level Language

When you write programs in Python, you do not have to worry about low-level details such as managing the memory used by your program, etc.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed such that it can work on) many platforms. You can use Python on Linux, Windows, FreeBSD, Mac OS, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE, Pocket PC and even Nokia

Series 60 mobile phones.

All your Python programs can work on most of these platforms without requiring any changes at all, *if* you are careful enough to avoid any system-specific features.

Interpreted

This requires a bit of explanation.

A program written in a compiled language is converted from the source language (such as C or C++) into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it immediately.

Python, on the other hand, does not need compilation to a binary. You just **run** the program directly from the source code. Internally, Python converts the source code into an intermediate format called bytecodes and then the interpreter just runs the bytecode. This makes Python easier to use since you do not have to worry about compiling the program, making sure the libraries are installed properly, etc. among other things. This also makes your Python programs more portable since you can just copy your Python program to a different computer and it just works.

Object-oriented

Python supports procedure-oriented programming as well as object-oriented programming.

In procedure-oriented languages, the program is built around procedures or functions which are nothing but reusable pieces of functionality. In object-oriented languages, the program is built around objects which combine data and functionality. Python has a very simplistic but powerful way of doing object-oriented programming (OOP).

Extensible

If you need a critical piece of code to run very fast or want to have some piece of algorithm to be not known publicly, you can write that part of your program in C or C++ and then use them from your Python program.

Embeddable

You can embed Python within your C/C++ programs to give 'scripting' capabilities for your program's users.

Extensive Libraries

The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, ftp, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces) using Tk, and other system-dependent functionality. Remember, the standard library is always available wherever Python is installed. This is called the *Batteries Included* philosophy of Python.

Besides the standard library, there are a huge number of high-quality libraries available for Python. See the [Python CheeseShop](#) for an official list of these libraries.

Summary

Python is indeed an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy.

Why not Perl?

If you did not know already, Perl is another extremely popular open source interpreted programming language.

If you have ever tried writing a large program in Perl, you would have answered this question yourself! In other words, Perl programs are easy when they are small and it excels at small programs and short scripts to "get the work done". However, they become unwieldy when you start writing larger programs or start implementing new algorithms.

In comparison with Perl, Python programs are simpler, clearer, and easier to write. Hence, they are more understandable and maintainable. I personally love Perl, but whenever I write a program, I always start thinking in terms of Python because the "Python way" of writing programs feels so natural.

The significant advantage that Perl has is its huge [CPAN](#) - the Comprehensive Perl Archive Network. As the name suggests, this is a humongous collection of Perl modules. It is mind-boggling because of its sheer size and depth - you can do virtually anything you can do with a computer using these modules. One of the reasons that Perl has more libraries than Python is that it has been around for a much longer time than Python.

The in-progress [Parrot virtual machine](#) is designed to run both the completely redesigned Perl 6 as well as Python and other interpreted languages like Ruby, PHP and Tcl. What this means to us is that *maybe* in future, we will be able to use Perl modules from Python which gives us the best of both worlds. We will just have to wait and see what happens.

What Programmers Say

You may find it interesting to read what many hackers and programmers have to say about Python.

Eric S Raymond

Eric S Raymond is the author of *The Cathedral and the Bazaar*, a famous essay on open source and Linux. He says that Python has become his [favorite programming language](#). This article was the real inspiration for my first brush with Python.

Bruce Eckel

Bruce Eckel is the author of the famous *Thinking in C++* and *Thinking in Java* books. He says that no language has made him more productive than Python. He says that Python is perhaps the only language that focuses on making things easier for the programmer. Read the [complete interview](#) for details.

Peter Norvig

Peter Norvig is a well-known Lisp hacker/author and is currently the Director of Search Quality at Google. He says that "Python has always been an integral part of Google".

You can actually verify this statement by looking at the [Google Jobs](#) page lists Python knowledge as one of the requirements for software engineers. Also, Guido van Rossum (the creator of Python) has now [joined Google](#) and gets to work on the open source version of Python during 50% of his time.

Installing Python

For Windows users

There are two choices of how to install Python on Windows:

- Visit the [python.org download](http://python.org/download) page and download the latest .exe file. The installation is just like any other Windows software.
 - Note: Do not uncheck any items when you are given the option during installation. Some of these components can be useful for you, such as IDLE.
- Download and install [ActivePython](#) - this is a Python installer provided by ActiveState. It has many goodies bundled such as PythonWin which will be useful for Windows users.

Note

ShowMeDo.com has a video that explains [how to download, install and run Python](#).

Using Python in the Windows command line

If you want to be able to use Python from the Windows command line, then you need to set the PATH variable appropriately.

For Windows 2000, XP, 2003, click on Control Panel -> System -> Advanced -> Environment Variables. In the 'System Variables' section, click on the variable named PATH, select 'Edit' and add ;C:\Python24 at the end. Make sure this is the correct folder name on your system.

For older versions of Windows, add the line `PATH=%PATH%;C:\Python24` to the file `C:\AUTOEXEC.BAT` and restart the system. For Windows NT, use the `AUTOEXEC.NT` file.

For Linux/BSD users

If you are using a Linux distribution such as Ubuntu, Fedora, Mandriva or {put your choice here}, or a BSD system such as FreeBSD, then you probably already have Python installed on your system.

To test if you have Python already installed on your Linux system, open a shell program (like konsole or gnome-terminal) and enter the command `python -V` (notice the capital V).

```
$ python -V
Python 2.4.2
```

Note

\$ is the prompt of the shell. It will be different for you depending on the settings in your system, hence I will indicate the prompt by just the \$ symbol.

If you see some version information like the one shown above, then you have Python installed already.

However, if you get a message like this:

```
$ python -V
bash: python: command not found
```

Then, you don't have Python installed. This is highly unlikely but possible. In this case, you have two ways of installing Python on your system:

- Install the binary packages using the package management software that comes with your system (such as apt-get in Ubuntu/Debian Linux, yum in Fedora Linux, urpmi in Mandriva Linux, pkg_add in FreeBSD, etc.) Note that you will need an active internet connection to use this method.
- Alternatively, you can download the binaries (such as .deb or .rpm) and then copy it to your computer and install it.
- You can compile python from the [source code](#) and install it. The compilation instructions are provided with the source code.

For Mac users

Python is installed by default on Mac OS X 10.3 and above. If you want to install a newer version of Python, use DarwinPorts:

- [Install DarwinPorts](#)
- Run `sudo port search python` to get the list of python software - as of this writing, Python 2.4.2 is the latest and is available in DarwinPorts as python24.
- Run `sudo port install python24`

For older versions of Mac OS, visit the [MacPython official downloads page](#), download the .dmg file specific to your Mac OS version, mount the disk image, and run the installer.

Summary

We will now assume that you have Python installed on your system.

Next, we will write our first Python program.

First Steps

Introduction

We will now see how to run a traditional 'Hello World' program in Python. This will teach you how to write, save and run Python programs.

There are two ways of using Python to run your program - using the interactive interpreter prompt or using a source file. We will now see how to use both the methods.

Using the interpreter prompt

There are two ways to get to the Python prompt - open a shell prompt and then start running the Python interpreter, or by directly starting the Python interpreter.

- Linux/BSD users can open a terminal/shell program such as `konsole` or `gnome-terminal`. Then, type `python` and press the enter key.
- Mac OS X users can access the shell by opening Finder → Applications → Utilities → Terminal. Then, type `python` and press the enter key.
- Windows XP users can open the shell by running (Start → Run) `cmd`. Then, type `python` and press the enter key.
 - Users of older versions of Windows can run `command`.
- To directly use the Python interpreter, we can use programs such as IDLE. Windows users can access it via the menu Start → Programs → Python 2.4 → IDLE (Python GUI).

Now type `print 'Hello World'` and press the enter key. You should see the words Hello World printed to the screen.

```
$ python
Python 2.4.2 (#1, Jan 20 2006, 21:12:19)
[GCC 4.0.0 20041026 (Apple Computer, Inc. build 4061)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello World'
Hello World
>>>
```

Note

The `>>>` sign is an indication that Python is waiting for us to enter our program.

Notice that Python gives you the output of the line immediately! What you just entered is a Python *statement*. We use `print` to, unsurprisingly, print any value that you supply to it. In this case, we are supplying the text `Hello World` and this is promptly printed to the screen.

How to quit the Python prompt

- Windows users - Press Ctrl-Z followed by enter key.
- Linux/BSD/Mac OS X users - Press Ctrl-D.

Choosing an Editor

Before we learn how to write Python programs in files, we need an editor to write in files. The choice of an editor is crucial. You have to choose an editor as you would choose a car you would

buy. A good editor will help you write Python programs easily, making your journey more comfortable and help you reach your destination (achieve your goal) in a much faster and safer way.

One of the basic requirements of a good editor is **syntax highlighting** where all the different parts of your Python program are colorized so that you can *see* your program and visualize its running.

Windows

Windows users can use [PythonWin](#). PythonWin provides syntax highlighting, good support for editing Python programs, and allows you to run programs from within PythonWin, among other things.

In particular, please *do not use Notepad* - it is a bad choice because it does not do syntax highlighting and importantly, it does not support indentation of the text.

Note : If you used the [ActivePython installer to install Python](#), PythonWin is already installed on your computer. You can open it by clicking on Start → Programs → ActiveState ActivePython 2.4 → PythonWin IDE.

Linux/BSD

If you are a beginner, you can use KWrite or GEdit programs.

If you are an experienced programmer, then you must be already using Vim or Emacs.

Mac OS X

Mac OS X users can use [TextMate](#) or [DrPython](#). [TextWrangler](#) is a free offering from BBEdit.

Vim or Emacs

[Vim](#) and [Emacs](#) are the two most powerful editors available and they run on all platforms - Windows, Mac OS X, Linux and BSD. If you are going to write a lot of Python code, I recommend that you learn one of these editors because they will make writing code so much easier for you. I personally use Vim.

More choices

To explore more choices of editors, see the comprehensive [list of Python editors](#). You can also choose an IDE(integrated development environment) from the [list of IDEs that support Python](#).

Again, please choose a proper editor - a good editor can make writing Python programs much easier.

[\[edit\]](#)

Using Source Files

There is a tradition that whenever you learn a new programming language, the first program that you write and run is the 'Hello World' program - all it does is just say 'Hello World' when you run it. As Simon Cozens (the author of the brilliant 'Beginning Perl' book) says, it is the *traditional incantation to the programming gods to help you learn the language better*.

Start your editor, type the following program and save it as `helloworld.py`.

```
print 'Hello World'
```

Run this program by opening a shell prompt and run the command `python helloworld.py`. If you are using PythonWin, use the menu File → Run.

The output is as shown below:

```
$ python helloworld.py
Hello World
```

If you see the output as shown above, congratulations! You have successfully run your first Python program.

In case you got an error, please type the above program *exactly* as shown above and try running the program again. Note that Python is case-sensitive i.e. `print` is different from `Print` - note the capital P. Also, ensure there are no spaces or tabs at the beginning of each line (we will see why this is important later).

How It Works

The `print statement` is supplied with the text `Hello World`, and it promptly prints it to the screen.

Note that we indicate text by surrounding it with quotes. We also refer to text as strings since they are a string of characters.

Comments

When we are writing programs, we might want to add some notes that will help us remember why we wrote the program in a certain manner, or add notes about some things that are on the todo list, etc. We can write these within our program itself using comments.

Comments are anything that is to the right of the `#` symbol (and the `#` is not inside a string).

For example, consider the following program.

```
# Start of the program
print 'Hello World' # The traditional first program
```

This program gives the same output as the previous program. The only difference is that we have added comments to our program source.

Comments are important because they help you to write down notes related to the programs that can be very helpful to readers of the program. It will also be helpful to you when you read your program, say, six months from now. So, please use comments liberally in your programs.

Remember to write comments that explain *what* your program is doing rather than *how* your program achieves it (which is explained by the code itself).

Executable Python programs

Note

This section is mainly of interest to Linux/BSD/Mac users. Windows users may skip this section.

In practice, Python programs have a special comment in the first line called the *shebang* line:

```
#!/usr/bin/env python
print 'Hello World'
```

This program is still the same 'Hello World' program but the first line here has special significance. Whenever the first two characters of the source file are `#!` followed by the location of a program, this tells your Linux/BSD/Mac OS X system that this program should be run with the interpreter

location specified in the shebang line when you run the program *directly*.

Note that you can always run the program on any platform by specifying the interpreter directly on the command line such as the command `python helloworld.py`. However, if we want to enable our programs to be freely usable like the built-in programs on our computer, we can use the shebang line to achieve that.

For the python location, we use `/usr/bin/env python` instead of specifying the exact location of the python command. This is to allow compatibility across different systems because some systems have python installed at `/usr/local/bin/python`, some have it at `/usr/bin/python` and so on. By using `env` to find the python command for us, we can ensure the program will run on different Unix-like systems.

To see how the shebang line is useful, let us give the program 'execute permission' using the `chmod` command. This tells us our system that this is not an ordinary file, but can be executed. Then, we actually *run* the program.

```
$ chmod a+x helloworld.py
$ ./helloworld.py
Hello World
```

The `chmod` command is used here to **change** the **mode** of the file by giving execute permission to **all** users of the system. Then, we execute the program directly by specifying the location of the source file. We use the `./` to indicate that the program is located in the current directory.

How does the system know how to run this source file? That's where the shebang line comes into the picture. All that the system does is look up the location of the interpreter specified in the shebang line and passes our file to that interpreter and voila, our program runs.

To make things more fun, you can rename the file to just `helloworld` and run it as `./helloworld` and it still works, since the system knows how to run this program.

You are now able to run the program as long as you know the exact path of the program, but what if you wanted to be able to run the program from anywhere? You can do that by storing the program in one of the directories listed in the `PATH` environment variable.

Whenever you run any program on your system, the system looks for that program in each of the directories listed in the `PATH` environment variable and then runs that program. We can make our program available everywhere by simply copying it to one of the directories listed in the `PATH` environment variable or by adding the directory of our program to the `PATH` environment variable.

```
# Copy program to a directory already present in PATH
$ echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/Users/swaroop/Code
$ cp helloworld.py /Users/swaroop/Code/helloworld
$ helloworld
Hello World
```

OR

```
# Add the directory to the PATH
$ pwd # print working directory
/Users/swaroop/Code
$ export PATH=$PATH:/Users/swaroop/Code # add directory to the PATH
$ echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/Users/swaroop/Code
$ helloworld
Hello World
```

We can display the value of the `PATH` environment variable using the `echo` command and by prefixing `$` to the name of the environment variable. The `$` indicates that we want to *retrieve* the value of the variable. We see that `/Users/swaroop/Code` is present in the `PATH`. Note that `swaroop` is my username on my Mac OS X system. There will be a similar `HOME` directory for your system with your username. We can copy our program to this directory and when we simply run `helloworld`, the system finds the program in the `PATH` and runs it.

In the second example, we are adding the directory to the `PATH`. We use `$PATH` to get the value of the environment variable and then set the value again by adding the directory path. Note that the list of directories are separated in the `PATH` variable by colons (`:`), so we add a semicolon as well. The `export` command specifies that the programs that will run in our current shell should be able to use this new `PATH` value.

An important thing to note is that by achieving all this, our program has become a part of the system just like the myriad commands that we have use. So, you can write programs to automate some of the routine work that you do and you can make it a part of your system.

Getting Help

If you need quick information about any function or statement in Python, then you can use the built-in `help` functionality. This is very useful especially when using the interpreter prompt. For example, run `help(str)` and this displays the help for the `str` class which is how Python stores **strings** (text) that you use in your program (Classes will be explained in detail in a separate chapter).

Press `q` to quit the help.

Similarly, you can obtain information about almost anything in Python. Run `help()` to learn more about `help` itself.

In case you need help for statements like `print`, then you need to set the `PYTHONDOCS` environment variable appropriately to the Python documentation. This can be done easily on Linux/BSD/Mac OS X using the `env` command. First, you will have to [download the HTML documentation](#) from the `python.org` website.

```
$ export PYTHONDOCS=/Users/swaroop/Documents/Python/Docs/
$ python
Python 2.4.2 (#1, Jan 20 2006, 21:12:19)
[GCC 4.0.0 20041026 (Apple Computer, Inc. build 4061)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> help('print')
```

Notice that we have to use quotes to specify `'print'` so that Python understands we want to fetch help about `print`, we do not want to print anything.

Summary

You should now be able to write, save and run Python programs with ease.

Now that you are a Python user, let's learn some more Python concepts in the next chapter.

Basics

Introduction

Just printing `Hello World` is not enough, is it? You want to do more than that - you want to take some input, manipulate it and get something out of it. We can achieve this using constants and variables.

Literal Constants

An example of a literal constant is a number like `5`, `1.23`, `9.25e-3` or a string like `'This is a string'` or `It's a string!`. It is called a literal because it is *literal* i.e. you always use its value literally. For example, the number `2` always represents itself and nothing else. It is a constant because its value cannot be changed. Hence, they are called as literal constants.

Numbers

Numbers in Python are of four types - integers, long integers, floating point and complex numbers.

- Examples of integers are `2`, `1025` which are whole numbers.
- Long integers are just bigger whole numbers.
- Examples of floating point numbers (or floats) are `3.23` and `52.3E-4`. The E notation indicates powers of 10. In this case, `52.3E-4` means $52.3 * 10^{-4}$.
- Examples of complex numbers are `(-5 + 4j)` and `(2.3 - 4.6j)`.

Strings

Strings are basically just a bunch of words. In more formal terms, a string is a *sequence* of *characters*.

The simplest way of specifying strings is using single quotes or double quotes.

Single Quotes

You can specify strings using single quotes such as `'Quote me on this'`. All white space i.e. spaces and tabs are preserved as-is.

Double Quotes

Strings in double quotes work exactly like single quotes. An example is `"What's your name?"`.

Note for C/C++ Programmers

There is no separate `char` data type in Python. There is no real need for it and it is likely you will not miss it.

Note for Perl/PHP Programmers

Remember that single-quoted strings and double-quoted strings are the same - they do not differ in any way.

Variables

Using just literal constants can soon become boring - we need some way of storing any information and manipulate them. This is where *variables* come into the picture. Variables are exactly what they mean - their value can vary i.e. you can store anything using a variable. Variables are just parts of your computer's memory where you can store some information. Unlike literal constants, you need some way of accessing these variables and hence you give them names. So, first we will see what kind of names we can give for variables.

Identifier naming

Variables are examples of identifiers. Identifiers are names given to identify *something*. There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (upper or lower case) or an underscore (`_`).
- The rest of the identifier can consist of letters (upper or lower case), underscore (`_`) or digits (0-9).
- Identifier names are case-sensitive. For example, `myname` and `myName` are **not** the same, they are two different identifier names.
- Examples of *valid* identifier names are `i`, `count`, `__my_name`, `name_23`, `a1b2_c3`.
- Examples of *invalid* identifier names are `2things`, `this is spaced out`, `my-name`.

Data Types

Variables can hold values of different types called **data types**. The basic types are numbers and strings, which we have already discussed. In later chapters, we will see how to create our own types using classes.

Objects

Remember, Python refers to *anything* used in a program as an *object*. Python is strongly object-oriented in the sense that everything is an object including numbers, strings, functions and classes.

Example of using Variables and Literal Constants

We will now see how to use variables along with literal constants. Save the following example and run the program.

How to write Python programs

Henceforth, the standard procedure to save and run a Python program is as follows:

1. Open your editor of choice.
2. Enter the program code given in the example.
3. Save it as the file with the file name mentioned in the comment in the second line of the program. Follow the convention of having all Python programs saved with the extension `.py`.
4. Run the interpreter with the command `python program.py` or use IDLE to run the programs. You can also use the executable `python` programs method as explained earlier.

```
#!/usr/bin/env python
# File name: var.py
```

```
i = 5
print i
i = i + 1
print i

s = 'This is a string.'
print s
```

Output

```
$ python var.py
5
6
This is a string.
```

How It Works

First, we assign the literal constant value 5 to the variable called `i` using the assignment operator `=`. This line is called a statement because it *states* that something should be done and in this case, we connect the identifier `i` to the value 5. Next, we output the value of `i` using the `print` statement which promptly prints the value of the variable to the screen.

Then, we add 1 to the value stored by `i` and then store the result value back in `i`. We print the values before and after the statement, and we confirm that the value of `i` is now 6. An important thing to notice is that the right hand side of the statement is evaluated first and then the resulting value is assigned to the variable name on the left hand side. That is why the statement `i = i + 1` works even though we have `i` on both sides of the assignment operator (`=`).

Similarly, we assign the literal string to the variable `s` and then print it.

Note for C/C++ Programmers

Variables are used by just assigning them a value. No declaration or data type definition is needed.

Logical and Physical Lines

A physical line is what *you see* when you write the program. A logical line is what *Python sees* as a single statement. Python implicitly assumes that each *physical line* corresponds to a *logical line*.

An example of a logical line is a statement like `print 'Hello World'` - this is a single statement, and this also corresponds to a physical line (as you would see in an editor).

Implicitly, Python encourages the use of a **single statement per line** which makes code more readable.

If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (`;`) which indicates the end of a logical line/statement.

For example,

```
i = 5
print i
```

is effectively the same as

```
i = 5;
```

```
print i;
```

and the same can be written as

```
i = 5; print i;
```

or even

```
i = 5; print i
```

However, I **strongly recommend** that you stick to **writing a single logical line in a single physical line**. Use more than one physical line for a single logical line only if the logical line is really long. The idea is to avoid the semicolon as far as possible since it leads to more readable code. In fact, I have *never* seen a semicolon in any Python program.

An example of writing a logical line spanning many physical lines is

```
s = 'This is a string. \  
This continues the string.'  
print s
```

Output

```
This is a string. This continues the string.
```

Notice the use of the backslash to indicate that the logical line continues in the next physical line. This is referred to as *explicit line joining*.

Similarly,

```
print \  
i
```

is the same as

```
print i
```

Sometimes, there is an implicit assumption where you do not need a backslash. This is the case where the logical line uses parentheses, square brackets or curly braces. This is called **implicit line joining**. You can see this in later chapters when we write programs using lists.

Indentation

Whitespace is important in Python. Actually, *whitespace at the beginning of the line is important*. This is called **indentation**. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together **must** have the same level of indentation. Each such set of statements is called a **block**. We will see examples of how blocks are important in later chapters.

One thing you should know right now is that incorrect indentation can give rise to errors.

For example,

```
i = 5      # Error! Notice a single space at the start of the line  
print i
```

Output

```
$ python
>>> i = 5
    File "<stdin>", line 1
      i = 5
      ^
SyntaxError: invalid syntax
```

Notice that there is a single space at the beginning of the first line. The error indicated by Python tells us that the *syntax* of the program is invalid i.e. the program does not follow the rules of the language. What this means to you is that *you cannot arbitrarily start new blocks of statements*, except for the main block which you have been using all along.

The situations where you can use new blocks will be detailed in later chapters such as the chapter on [control flow](#).

How to indent

Do **not** use a mixture of tabs and spaces for the indentation as it does not work reliably across different platforms. The official recommended standard is *four spaces* for each level of indentation. You can also use tabs instead. More importantly, choose one style of indentation and use it consistently.

Note for Vim users

If you use Vim, you can enable tabs to be automatically converted to four spaces so that you do not have to enter four spaces each time yourself. You can also enable smart indentation to allow automatic indentation for subsequent statements of the same block.

To enable these features, you can put the following lines in the `.vimrc` in your home directory:

```
" Indentation
set smartindent

" Tabs to 4-spaces
set shiftwidth=4
set tabstop=4
set expandtab
set smarttab
```

More information is available in [Vim Tip #12](#).

Summary

We have discussed many nitty-gritty details of the basics of Python. Be sure to be comfortable with what you have read in this chapter as it lays the foundation of what we discuss in subsequent chapters.

Next, we move on to learning about operators and expressions.

Operators and Expressions

Introduction

Most statements (logical lines) that you write will contain *expressions*.

Consider the statement `i = 2 + 3`. We can break this down into three pieces - the left hand side, the assignment operator (=) and the right hand side.

The left hand side here is the variable name `i`. We can have only a single variable on the left hand side because we are assigning a value and only variables can hold values.

The right hand side consists of the *expression* `2 + 3` which itself can be broken down to three parts - the *operand* `2`, the *operator* `+` and the second operand `3`.

Operators are functionality that do something and can be represented by symbols such as `+` or by special keywords. Operators require some data to operate on and such data are called *operands*.

Operators

We will briefly take a look at the list of operators and their usage.

Note on testing the examples

You can evaluate the expressions given in the examples using the interactive interpreter prompt. For example, to test the expression `2 + 3`:

```
>>> 2 + 3
5
```

Operator	Name	Explanation	Examples
<code>+</code>	Plus	Add the two objects	<code>3 + 5</code> gives 8. <code>'a' + 'b'</code> gives <code>'ab'</code> .
<code>-</code>	Minus	Either gives a negative number or gives the subtraction of one number from another	<code>-5.2</code> gives a negative number. <code>50 - 24</code> gives the value 26.
<code>*</code>	Multiplication	Gives the multiplication of two numbers or returns the string repeated as many times as the following number.	<code>2 * 3</code> gives 6. <code>'la' * 3</code> gives <code>'lalala'</code> .
<code>**</code>	Power	Returns x to the power of y	<code>3 ** 4</code> gives 81 (= <code>3 * 3 * 3 *</code>

			3)
/	Division	Divide x by y	4 / 3 gives 1 (division of integers gives an integer). 4.0 / 3 or 4 / 3.0 gives 1.3333333333333333.
//	Floor division	Returns the floor of the quotient	4 // 3.0 gives 1.0.
%	Modulo	Returns the remainder of the division	8 % 3 gives 2. -25.5 % 2.25 gives 1.5.
<<	Left shift	Shifts the bits of the number to the left by the number of bits specified. Each number is represented in memory by bits or binary digits i.e. 0s and 1s.	2 << 1 gives 4 since 2 is represented in binary as 10. Left shifting this by 1 gives 100 which represents the decimal 4.
>>	Right shift	Shifts the bits of the number to the right by the number of bits specified.	11 >> 1 gives 5 since 11 is represented in binary as 1011, right shifting by 1 gives 101 which represents the decimal 5.
&	Bitwise AND	Bitwise AND of the numbers i.e. corresponding bits of the numbers are ANDed.	5 & 3 gives 1.
	Bitwise OR	Bitwise OR of the numbers i.e. corresponding bits of the numbers are ORed.	5 3 gives 7.
~	Bitwise invert	Bitwise inversion of x is $-(x+1)$	~5 gives -6.
<	Less than	Returns whether x is less than y. All comparison operators	5 < 3 gives 0 i.e. False. 3 < 5 gives 1 i.e. True. Comparison operators can

		return 1 for true and 0 for false. This is equivalent to the special names <code>True</code> and <code>False</code> . Note the capitalization of these names.	be chained arbitrarily, for example, <code>3 < 5 < 7</code> gives <code>True</code> .
<code>></code>	Greater than	Returns whether x is greater than y	<code>5 > 3</code> gives <code>True</code> . If both operands are numbers, they are first converted to a common type of number. If x and y are not numbers and are not objects that can be compared, then it returns <code>False</code> .
<code><=</code>	Less than or equal to	Returns whether x is less than or equal to y	<code>x = 3; y = 6; x <= y</code> gives <code>True</code> .
<code>>=</code>	Greater than or equal to	Returns whether x is greater than or equal to y	<code>x = 4; y = 3; x >= y</code> gives <code>True</code> .
<code>==</code>	Equal to	Compares if the two objects are equal in value	<code>x = 2; y = 2; x == y</code> returns <code>True</code> . <code>x = 'abc'; y = 'Abc'; x == y</code> returns <code>False</code> .
<code>!=</code>	Not equal to	Compares if the two objects are not equal in value	<code>x = 2; y = 3; x != y</code> returns <code>True</code> .
<code>not</code>	Boolean NOT	If x is <code>None</code> , 0, <code>False</code> or an empty string, then <code>not x</code> returns <code>True</code> . Otherwise, <code>not x</code> returns <code>False</code> .	<code>x = True; not x</code> returns <code>False</code> . <code>x = ; not x</code> returns <code>True</code> .
<code>and</code>	Boolean AND	If x is <code>False</code> , x and y returns <code>False</code> , otherwise it returns the evaluation of y.	<code>x = False; y = True; x and y</code> returns <code>False</code> since x is <code>False</code> . In this case, Python will not evaluate y since it knows that the value of x and y will always be <code>False</code> irrespective of the value of y because the first operand x is <code>False</code> . This is called <i>short-circuit evaluation</i> .

or	Boolean OR	If <code>x</code> is <code>True</code> , it returns <code>True</code> , else it returns the evaluation of <code>y</code> .	<code>x = True; y = False; x or y</code> returns <code>True</code> . Short-circuit evaluation applies here also.
----	------------	--	--

Operator Precedence

Consider the expression `2 + 3 * 4`. Is the addition done first or the multiplication? According to the [BODMAS rule](#), we have to do multiplication first and then addition i.e. the multiplication operator has higher precedence than the addition operator.

The following table gives the operator precedence table for Python, from the lowest precedence (least binding) to the highest precedence (most binding). This means that in a given expression, Python will first evaluate the operators lower in the table than the operators listed higher in the table.

The following table is the same as the one in the Python reference manual. It is provided for the sake of completeness. You are advised to use parentheses for grouping of operators and operands to explicitly specify the calculations to be as clear as possible, and avoiding referring this table. For example, `2 + (3 * 4)` is clearer than `2 + 3 * 4`. However, as with everything else, parentheses should be used judiciously and should not be redundant such as in `(2 + 3) + 4`. The parentheses in this example is redundant because the `+` operator is a left-to-right operator and the operation is clear that `2 + 3` will be evaluated first.

Operator	Description
<code>lambda</code>	Lambda expression
<code>or</code>	Boolean OR
<code>and</code>	Boolean AND
<code>not x</code>	Boolean NOT
<code>in, not in</code>	Membership tests
<code>is, is not</code>	Identity tests
<code><, <=, >, >=, !=, ==</code>	Comparisons
	Bitwise OR
<code>^</code>	Bitwise XOR
<code>&</code>	Bitwise AND
<code><<, >></code>	Shifts
<code>+, -</code>	Addition, Subtraction
<code>*, /, %</code>	Multiplication, Division, Remainder

+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation
x.attribute	Attribute reference
x[index]	Subscription
x[start:stop]	Slicing
f(arguments ...)	Function call
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key : datum, ...}	Dictionary display
`expressions ...`	String conversion

The operators that have not been discussed already will be explained in later chapters.

Operators with the **same precedence** are listed in the same row. For example, + and – have the same precedence.

Order of Evaluation

By default, the operator precedence table decides which operators are evaluated before others. However, if you want to change the order in which the operations are executed, you can use parentheses. For example, in the expression $2 + 3 * 4$, if you want the addition to be evaluated before the multiplication, you can rewrite the expression as $(2 + 3) * 4$.

Associativity

Operators are usually associated from left to right i.e. operators with the same precedence are evaluated in a left to right manner. For example, $2 + 3 + 4$ is evaluated as $(2 + 3) + 4$. Some operators like the assignment operator have right to left associativity i.e. $a = b = c$ is treated as $a = (b = c)$.

Expressions

Let us see an example of how to use expressions.

```
#!/usr/bin/env python
# File name: expression.py

length = 5
breadth = 2

area = length * breadth
print 'Area is', area
print 'Perimeter is', 2 * (length + breadth)
```

Output

```
$ python expression.py
Area is 10
Perimeter is 14
```

How It Works

The length and breadth of a rectangle are stored in variables of the same name. We use these to calculate the area and perimeter of the rectangle with the help of expressions. We store the result of the expression `length * breadth` in the variable `area` and then print it using the `print` statement. In the second case, we directly use the value of the expression `2 * (length + breadth)` in the `print` statement.

Also, notice how Python *pretty-prints* the output. Even though we have not specified a space between `'Area is'` and the variable `area`, Python puts it for us so that we get a nice output and the program is also "clean".

Summary

We have seen how to use operators and operands in expressions. These are the basic building blocks of every Python program.

Next, we will see how to make use of these expressions using statements.

Control Flow

Introduction

In the programs that we have written till now, the program has always been a series of statements which Python faithfully executes them in the same order. What if you wanted to *control the flow* of how it works? How do you make the program take decisions and do different things in different situations? For example, saying 'Good Morning' or 'Good Evening' depending on the time of the day.

As you might have guessed, we can do this using control flow statements. There are three control flow statements in Python - the `if`, `for` and `while` statements.

The if statement

The `if` statement is used to check a condition and *if* the condition is `True`, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional.

We can chain together related `if` statements using the shortcut name `elif` instead of `else if`.

```
#!/usr/bin/env python
# File name: if.py

number = 23
guess = int(raw_input('Enter an integer : '))

if guess == number:
    print 'Congratulations, you guessed it.'           # New block starts here
    print '(but you do not win any prizes!)'         # New block ends here
elif guess < number:
    print 'No, it is higher than that.'              # Another block
    # You can do whatever you want in a block ...
else:
    print 'No, it is lower than that.'
    # you must have guess > number to reach this block

print 'Done'
# This last statement is separate from the if statement,
# and since it is present in the main block, it is always executed.
```

Output

```
$ python if.py
Enter an integer : 50
No, it is lower than that.
Done
```

```
$ python if.py
Enter an integer : 22
No, it is higher than that.
Done
```

```
$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

How It Works

In this program, we take guesses from the user and check if it is the number that we have. We set the variable `number` to any integer we want, say 23. Then, we take the user's guess using the `raw_input()` function. It would be easier to just use `input()` though.

Functions are just reusable pieces of programs. We will learn more about functions in the [next chapter](#).

`raw_input()` is a part of the Python language, so it is always available for us to use. If we supply a string to the `raw_input()` function, it prints the string to the screen and then waits for input from the user. Once we enter something and press enter/return key, the function returns that text to us. We then convert this string to an integer using `int` and then store the resulting integer value using the variable `guess`.

The `int` is actually a class but all you need to know right now is that you can use it to convert a string to an integer, assuming the string contains a valid integer representation.

Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message. Notice that we use indentation levels to tell Python which statements belong to which block. This is why indentation is so important in Python. I hope you are sticking to the *four spaces per indentation level* rule. Are you?

Notice how the `if` statement contains a colon (`:`) at the end - this tells Python that a block of statement follows.

Then, we check if the guess is less than the number and if so, we inform the user to guess higher than that.

We have then used the `elif` clause to combine two related `if else-if else` statements into one combined `if-elif-else` statement. This makes the program easier to read and reduces the amount of indentation required.

The `elif` and `else` statements must also have a colon at the end of the logical line followed by their corresponding block of statements, with proper indentation.

You can have another `if` statement inside the `if`-block, `else`-block, or `elif`-block of an `if` statement - this is called a *nested* `if`-statement. As far as Python is concerned, it is just another statement within the `if`-block.

Remember that the `elif` and `else` parts are optional. A minimal valid `if` statement is

```
if True:
    print 'Yes, it is true'
```

After Python has finished executing the complete `if` statement along with the associated `elif` and `else` clauses, it moves on to the next statement in the block containing the `if` statement. In this case, it is the main block where the execution of the program starts and the statement after the `if` statement is the `print 'Done'` statement. After this statement, Python sees the end of the program and finishes.

Although this is a very simple program, I have been pointing out a lot of things that you should notice even in this simple program. Many of the nuances are pretty straightforward to understand and requires you to become aware of these initially, but once you become comfortable, it will feel natural to you.

Note for C/C++ programmers

There is no *switch* statement in Python. You can use an `if-elif-else` statement to do the

same thing.

The while statement

The `while` statement allows you to repeatedly execute a block of statements as long as a condition is `True`. A `while` statement is an example of what is called a *looping* statement, where it repeatedly executes (*loops*) a block of statements until a condition is met.

A `while` statement can have an optional `else` clause attached.

```
#!/usr/bin/env python
# File name: while.py

number = 23
running = True

while running:
    guess = int(raw_input('Enter an integer : '))

    if guess == number:
        print 'Congratulations, you guessed it.'
        running = False      # this causes the while loop to stop
    elif guess < number:
        print 'No, it is higher than that.'
    else:
        print 'No, it is lower than that.'
else:
    print 'The while loop is over'

print 'Done'
```

Output

```
$ python while.py
Enter an integer : 50
No, it is lower than that.
Enter an integer : 22
No, it is higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over
Done
```

How It Works

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly execute the program for each guess, as we have done previously. This aptly demonstrates the use of the `while` statement.

We have moved the `raw_input` and `if` statements to inside the `while` loop. First, we set the `running` variable to `True`, then we run the `while` loop. The `while` loop checks if the `running` variable is `True`, which is always correct the first time the `while` loop is executed. Then, the *while-block* is executed. After the block is executed, the condition is checked again i.e. the `running` variable is checked if it is `True`. If `True`, we execute the `while-block` again, else we continue to execute the optional `else-block` and then the next statement in the block containing the `while` statement.

The `else-block` is executed when the `while` loop condition becomes `False` - this may even be the first

time that a condition is checked. There is only one situation where the `else`-block is not executed, and that is when we use the `break` statement to stop the while loop. We will learn about the [break statement](#) later in this chapter.

Note for C/C++ Programmers

Remember that you can have an `else`-clause associated with a while loop. The `else`-clause is not executed if the `break` statement is used to stop the while loop.

The for loop

The `for...in` statement is another looping statement which *iterates* over a *sequence* of objects i.e. goes through each item in a sequence. We will learn more about [sequences](#) in a later chapter. What you need to know right now is that a sequence is just an ordered collection of items.

```
#!/usr/bin/env python
# File name: for.py

for i in range(1, 5):
    print i
```

Output

```
$ python for.py
1
2
3
4
```

How It Works

In this program, we are printing a *sequence* of numbers. We generate this sequence using the built-in `range` function.

If we supply two numbers to the `range` function, it returns a sequence of numbers starting from the first number and up to the second number. For example, `range(1, 5)` gives the sequence `[1, 2, 3, 4]`. The range extends *up to* the second number, it does *not* include the second number.

The `for` loop then iterates over this sequence. So, `for i in range(1, 5)` is equivalent to `for i in [1, 2, 3, 4]`. Each item in the sequence is assigned to the variable `i`, one at a time, and then the `for`-block is executed for each value. In this case, the block of statements just prints the value of `i` to the screen.

The `for...in` loop works for a sequence of any kind of objects. It just so happens that we have all numbers in the sequence in our example.

An `else` clause is optional for the `for` loop and is executed normally except when the `break` statement is used to stop the `for` loop.

Note for C/C++/Java/C# Programmers

The Python `for` loop is different from the traditional `for` loop where we use counters explicitly. Instead, the Python `for` loop simply iterates over a sequence of objects.

C# programmers will note that the `for` loop in Python is similar to the `foreach` loop in C#. Java programmers will note that this is similar to `for (int i: IntArray)` syntax in Java 1.5. Importantly, Python does not have restrictions on the type of the objects in the sequence - the objects could be of different types as well.

The break statement

The `break` statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become `False` or the sequence of items has not been completely iterated over.

When the `break` statement is used, the corresponding `else` clause of the loop statement is *not* executed.

```
#!/usr/bin/env python
# File name: break.py

while True:
    s = raw_input('Enter something : ') #underscore between raw _ input
    if s == 'quit':
        break
    print 'length of the string is', len(s)
print 'Done'
```

Output

```
$ python break.py
Enter something : Programming is fun
length of the string is 18
Enter something : When the work is done
length of the string is 21
Enter something : if you wanna make your work also fun:
length of the string is 37
Enter something :         use Python!
length of the string is 12
Enter something : quit
Done
```

How It Works

In this program, we repeatedly take the user's input and print the length of the input each time. We provide a special condition to stop the program by checking if the user's input is `'quit'`. We stop the program by *breaking* out of the loop.

The length of the input string can be measured using the built-in `len` function. See `help(len)` for details. Interestingly, `len` works for any sequence of objects, in this case it happens to be a sequence of characters i.e. a string.

Remember that the `break` statement can be used with the `for` loop as well.

The continue statement

The `continue` statement is used to tell Python to skip the rest of the statements in the current block and *continue* to the next iteration of the loop.

```
#!/usr/bin/env python
# File name: continue.py
```

```
while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        continue
    print 'Input is of sufficient length'
    # Do other kinds of processing here ...
```

Output

```
$ python continue.py
Enter something : abc
Input is of sufficient length
Enter something : abcd
Input is of sufficient length
Enter something : quit
```

How It Works

In this program, we accept input from the user, but we process only those inputs that are at least 3 characters long. So, we use the built-in `len` function to find out the length of the input and if the length is less than 3, we skip the rest of the block by using the `continue` statement. Otherwise, the rest of the statements in the `while`-block is executed.

Note that the `continue` statement works with the `for` loop as well.

Summary

We have seen how to use the three control flow statements - `if`, `while` and `for` along with their associated `break` and `continue` statements. We will use these statements often and hence, becoming comfortable with these statements is essential.

Next, we will see how to create and use functions.

Functions

Introduction

Functions are reusable pieces of programs. They allow you to give a name to a block of statements and you can run that block using that name any number of times and anywhere in your program. This is known as *calling* the function. We have already used many built-in functions such as `len` and `range`.

Functions are **defined** using the `def` keyword. This is followed by an identifier name for the function followed by a pair of parentheses which may enclose some names of variables and the line ends with a colon. This is followed by a block of statements that form the body of the function.

```
#!/usr/bin/env python
# File name: function1.py

def say_hello():
    print 'Hello World!' # body of the function

say_hello() # calling the function
```

Output

```
$ python function1.py
Hello World!
```

How It Works

We define a function called `say_hello` using the syntax as explained above. The function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it during a call and get back corresponding results.

We call the function by using parentheses, and it just runs. After the function finishes running, Python moves on to the next statement in the block containing the function call.

Function parameters

A function can take parameters which are values you can supply to any function during its call so that the function can *do* something utilizing those values and return an appropriate result. These parameters are just like variables except that the values of these variables are defined when we call the function and are not assigned values within the body of the function.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the arguments in the same order. Note the terminology used - the names given in the definition function are called *parameters* whereas the values you supply in the function call are called *arguments*.

```
#!/usr/bin/env python
# File name: param.py

def print_max(x, y):
    if x > y:
        print x, 'is maximum'
    else:
        print y, 'is maximum'
```

```
print_max(3, 4) # directly give literal constants as arguments

a = 5
b = 7

print_max(a, b)      # pass in variables as arguments
```

Output

```
$ python param.py
4 is maximum
7 is maximum
```

How It Works

We define a function called `print_max` where we take two parameters called `x` and `y`. We find out the bigger number of the two using a simple `if..else` statement and then print the bigger number.

In the first usage of the `print_max` function, we directly supply the arguments as literal constants. In the second usage, we call the function by giving variable names. Notice that Python treats both the cases in the same manner.

The call `print_max(a, b)` causes the value of argument `a` to be assigned to the parameter `x` and the value of the argument `b` assigned to the parameter `y`.

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names outside the function - i.e. variable names are *local* to the function. This is called the *scope* of the variable. All variables have the scope of the block they are declared in, starting from the point of definition of the variable.

```
#!/usr/bin/env python
# File name: local.py

def func(x):
    print 'x is', x
    x = 2
    print 'Changed local x to', x

x = 50
func(x)
print 'x is still', x
```

Output

```
$ python local.py
x is 50
Changed local x to 2
x is still 50
```

How It Works

In the function, in the first statement we use the *value* of the variable `x`, Python uses the value of the parameter declared in the function. Next, we assign the value 2 to `x`. The name `x` is local to our

function, so when we change the value, the `x` defined in the main block remains unaffected. We can confirm this in the last `print` statement where the value of `x` in the main block has not changed.

Namespace

You can imagine a function to be a bag within which you have declared variables. The variables exist only within that bag - i.e. the name exists within that space and hence functions are also *namespaces*.

The global statement

If you want to assign a value to a name defined outside the function, then you have to tell Python that the name is not local, but it is *global*. We do this by using the `global` statement. It is not possible to assign a value to a variable defined outside a function without using the `global` statement.

Although, you *can* use the values of such variables defined outside the function, assuming there is no variable with the same name defined within the function. However, this is not encouraged since it can be unclear to the reader of the program regarding where that variable's definition is located. Using the `global` statement makes it clear that the variable is defined in an outer block.

```
#!/usr/bin/env python
# File name: global.py

def func():
    global x

    print 'x is', x
    x = 2
    print 'Changed global x to', x

x = 50
func()
print 'Value of x is', x
```

Output

```
$ python global.py
x is 50
Changed global x to 2
Value of x is 2
```

How It Works

The `global` statement is used to declare that `x` is a global variable - hence, when we assign a value to `x` within the function, that change is reflected when we use the value of `x` in the main block.

You can specify more than one global variable using the same `global` statement. For example, `global x, y` declares that both `x` and `y` are global variables.

Default argument values

For some functions, you may want to mark some of its parameters as *optional* and use default values if the user does not want to provide values for such parameters. This is done with the help of

default argument values. You can specify default argument values for parameters by following the parameter name in the function definition with the assignment operator (=) followed by the default value.

Note that the default argument value should be a constant. More precisely, the default argument value should be **immutable**, this is explained in detail in later chapters.

```
#!/usr/bin/env python
# File name: default.py

def say(message, times = 1):
    print message * times

say('Hello')
say('World', 5)
```

Output

```
$ python default.py
Hello
WorldWorldWorldWorldWorld
```

How It Works

The function named `say` is used to print a string as many times as specified. If we do not supply a value on how many times we want it to be printed, the function just prints it once. We achieve this by specifying a default argument value of 1 to the parameter `times`.

In the first call of `say`, we supply only the text and it prints that string once. In the second call of `say`, we supply both the string and an argument 5 which results in the message being printed 5 times.

It is important to note that only those parameters which are at the end of the parameter list can be given default argument values. You cannot have a parameter with a default argument value before a parameter without a default argument value. This is because the values are assigned to the parameters by position. For example, `def func(a, b=5)` is valid, but `def func(a=5, b)` is not valid.

Keyword arguments

If you have some functions with many parameters and want to specify only some of them, then you can give values for such parameters by naming the parameters - this is called *keyword arguments*; we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* - one, using the function is easier since we do not have to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

```
#!/usr/bin/env python
# File name: keyword.py

def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output

```
$ python keyword.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

How It Works

The function named `func` has one parameter without a default argument value, followed by two parameters with default argument values.

In the first usage, `func(3, 7)`, the parameter `a` gets the value 3, the parameter `b` gets the value 7, and `c` gets the value of its default value 10.

In the second usage, `func(25, c=24)`, the parameter `a` gets the value of the first argument 25, the parameter `b` gets the default value of 5 and the parameter `c` gets the value 24 by the keyword argument value.

In the third usage, `func(c=50, a=100)`, we use keyword arguments to specify the value of `a` and `c`. Notice that we are specifying value for `c` before that for `a` even though `a` is defined earlier in the parameter list in the function definition.

The return statement

The `return` statement is used to *return* from a function (i.e. break out from a function). We can optionally *return a value* from the function as well.

```
#!/usr/bin/env python
# File name: return.py

def maximum(x, y):
    if x > y:
        return x
    else:
        return y

print maximum(2, 3)
```

Output

```
$ python return.py
3
```

How It Works

The `maximum` function returns the maximum of the two parameters. In this case, two numbers are supplied to the function call. It uses a simple `if..else` statement to find the bigger number and then *returns* that value.

Note that a `return` statement without a value is equivalent to `return None`, which is implicitly executed by every Python function if the function completes normally without using any explicit `return` statement. `None` is a special type in Python that represents nothingness. For example, if a variable has a value of `None`, it means that the variable has no value attached to it.

For example, if we run `print some_function()` where the function `some_function` does

not use the return statement, we can see the value None printed.

```
>>> def some_function():
...     pass
...
>>> print some_function()
None
>>>
```

The `pass` statement is used to indicate an empty block of statements.

DocStrings

Python has a nifty feature called *documentation strings*, or *docstrings* for short. DocStrings are an important tool that you should make use of, to document the program better and make it easy to understand. We can even get back the docstring when the program is running.

```
#!/usr/bin/env python
# File name: docstrings.py

def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers or strings containing integers.'''

    x = int(x)          # convert to integers, if possible
    y = int(y)

    if x > y:
        print x, 'is maximum'
    else:
        print y, 'is maximum'

print_max(3, 5)
print print_max.__doc__
```

Output

```
$ python docstrings.py
5 is maximum
Prints the maximum of two numbers.
```

```
    The two values must be integers or strings containing integers.
```

How It Works

A string on the first logical line of a function is the *docstring* for that function. Note that docstrings also apply to modules and classes, which we will learn about in the following chapters.

The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. The first line is usually a brief one-liner explanation about the function. If a longer explanation is to be included, the first line is followed by a blank line followed by the actual explanation from the third line onwards. You are *strongly advised* to follow this convention for all your docstrings for all your non-trivial functions.

We can access the docstring of the `print_max` function using the `__doc__` (notice the double underscores) attribute of (i.e. identifier belonging to) the function. We will learn more about [attributes](#) in the chapter on object oriented programming. Just recollect that Python treats *everything* as an object, and when a function has a docstring, it is part of the function which can be accessed by

its attribute `__doc__`.

We have already used the `help()` in Python, and so you have already seen the usage of docstrings. What `help()` does is just fetch the docstring i.e. the `__doc__` attribute of that function and displays it in a neat manner for you. You can try it out on the function above, just include `help(print_max)` in your program. Remember to press `q` to quit the help display.

Automated tools can retrieve the documentation from your program in this manner. Therefore, you should always write docstrings for any non-trivial function. The `pydoc` command that comes with your Python distribution works similar to `help()` using docstrings.

Summary

We have seen many aspects of functions but we have not covered all aspects of it. However, we have covered most aspects that you will use on an everyday basis.

Next, we will see how to use and create Python modules.

Modules

Introduction

You have seen how you can reuse code in your program by using functions. What if you wanted to reuse a number of common functions in different programs that you want to write? Yes, modules is the answer. A module is basically a file containing functions and variables that you have defined. To reuse the module in other programs/modules, we can *import* the module or some parts of the module, and reuse the functionality provided by that module.

Modules are also *namespaces* - the functions and variables defined within a module exist within the module (i.e. within that namespace) only.

Modules must have an extension of `.py` so that it can be imported in other Python programs/modules.

First, we will see how to use modules that are part of the standard library.

```
#!/usr/bin/env python
# File name: using_sys.py

import sys

print 'The command line arguments are:'
for i in sys.argv:
    print i

print '\nThe PYTHONPATH is', sys.path
```

Output

```
$ python using_sys.py we are arguments
The command line arguments are:
using_sys.py
we
are
arguments
The PYTHONPATH is
['/opt/local/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4',
'/opt/local/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/plat-
darwin',
'/opt/local/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/plat-
mac',
'/opt/local/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/plat-
mac/lib-scriptpackages',
'/opt/local/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/lib-
tk',
'/opt/local/lib/python2.4/lib-dynload',
'/opt/local/Library/Frameworks/Python.framework/Versions/2.4/lib/python2.4/site-
packages',
'/opt/local/lib/python2.4/site-packages',
'/Users/swaroopch/Library/Python/2.4/site-packages']
```

How It Works

First, we *import* the `sys` module into our program using the `import` statement. Basically, this tells Python that we want to use this module and we should bring that name into our program so that we can use the module. The `sys` module contains functionality related to the Python interpreter and its

environment. The name 'sys' is short for 'system'.

When Python executes the `import sys` statement, it looks for the corresponding Python module in the directories listed in the `sys.path` variable. If the module/library is found, then the statements in the main block of the `sys` module are executed and then the module is made *available* for us to use in our current program. Note that the initialization of that module is done only the first time we import that module in our program. Subsequent imports of the same module doesn't do anything.

The `argv` variable in the `sys` module is referred to using the dotted notation - `sys.argv` tells Python that we want to use the `argv` variable defined in the `sys` module. An advantage of this approach is that the name does not clash with any `argv` defined within our program. Also, it indicates clearly that this name is defined in the `sys` module namespace.

The `sys.argv` variable is a *list* of strings (lists are explained in the [next chapter](#)). Specifically, the `sys.argv` is a list of *command line arguments* i.e. the arguments passed to your program using the command line.

If you are using an IDE to write and run these programs, look for a way to specify command line arguments to the program in the menus. For example, in PythonWin, you can add text to the "Arguments" box when you click on File → Run.

Here, we execute `python using_sys.py we are arguments` and the `sys.argv` list contains the arguments we have specified following the actual `python` command itself. The name of the program running is always the first argument in the `sys.argv` list. So, in this case we will have `using_sys.py` as `sys.argv[0]`, `we` as `sys.argv[1]`, `are` as `sys.argv[2]`, and `arguments` as `sys.argv[3]`. Notice that Python starts counting from 0 and not 1.

Byte-compiled .pyc files

Importing a module is a costly affair. So, Python does some tricks to make it faster. Remember we discussed that Python converts the source code into an intermediate form called *bytecodes* and then translates that into the native language of the computer. The first time you import a module, Python stores those bytecodes as `.pyc` files. For example, if you run `import mymodule`, you might find a `mymodule.pyc` next to the `mymodule.py` that you have written. The next time you run `import mymodule` in any program, Python can pick up the `.pyc` file instead of the `.py` file and hence the import will be much faster. Python is smart enough to recreate the `.pyc` file if the `.py` module has been updated.

The byte-compiled files are platform-independent. Also, if Python does not have permission to create these `.pyc` files, the files are not created and the import will proceed like it is the first time.

The from...import statement

In the previous example, if you wanted to directly import the name `argv` into your module/program to avoid typing `sys.argv` every time, then you can use the `from sys import argv` statement.

If you want to import all the available names used in the `sys` module, then you can use `from sys import *`. This statement works for any module.

In general, it is advised *not* to use the `from...import` version because it will be difficult for the reader of the program to find out where the name has come from, and also it may cause name clashes with the names you have declared in the current module.

A module's `__name__`

Every module has a name and the functionality within a module can find out the name of its container module.

This is specially handy in one particular situation - Remember that when we import a module, all the statements in the main block of the module are executed and then the name is imported into our module (or *namespace*). What if we wanted to run a set of statements only if the module was used by itself and not when it was imported from another program. This can be achieved using the `__name__` attribute of the module.

```
#!/usr/bin/env python
# File name: name.py

if __name__ == '__main__':
    print 'This program is being run by itself.'
    print 'I can do other stuff here.'
else:
    print 'I am being imported from another module.'
```

Output

```
$ python name.py
This program is being run by itself.
I can do other stuff here.

$ python
>>> import name
I am being imported from another module.
```

How It Works

Every Python module has its `__name__` defined and if this is set to `__main__`, then it implies that the module is being run standalone by the user and we can do corresponding actions, else it is being imported by another module.

Making your own modules

Creating your own modules is easy, you have been doing it all along! Every Python program is also a module. You just have to make sure it has a `.py` file name extension.

```
#!/usr/bin/env python
# File name: mymodule.py

def sayhi():
    print 'Hi, this is', __name__, 'speaking'

version = '0.1'

# End of mymodule.py
```

The above is a sample *module*. As you can see, there is nothing particularly special about this module compared to the usual Python program that we have been writing. We will next see how to import this module into other programs.

Remember that a module has to be in one of the directories listed in the `sys.path` list, so that Python knows where to get this module from. The simplest way is to put both the `mymodule.py` and the program that is going to use this module in the same directory and Python will understand,

since the current directory is also part of the `sys.path` list.

```
#!/usr/bin/env python
# File name: mymodule_demo.py

import mymodule

mymodule.sayhi()
print 'Version', mymodule.version

# End of mymodule_demo.py
```

Output

```
$ python mymodule_demo.py
Hi, this is mymodule speaking
Version 0.1
```

How It Works

When we `import mymodule`, Python finds the module in the same directory and then runs the main block of that module and brings the `mymodule` name into our namespace.

Next, we use the dotted notation to access members of the module. This is what we called *accessing attributes*.

Also, notice the usage of the `__name__` special variable.

Here is a version utilising the `from..import` syntax:

```
#!/usr/bin/env python
# File name: mymodule_demo2.py

from mymodule import sayhi, version
# Alternative: from mymodule import *

sayhi()
print 'Version', version

# End of mymodule_demo2.py
```

The output of the above program is the same as the output of the previous program.

The `dir()` function

The built-in `dir()` function can be used to list the identifiers that a module defines. The identifiers are the functions, classes and variables defined in that module.

When you supply a module name to the `dir()` function, it returns the list of names defined in that module. When no argument is applied to it, it returns the list of names defined in the current module.

In fact, the `dir` function can be used for any object, not just modules.

```
$ python
>>> import mymodule
>>> dir(mymodule)
['__builtins__', '__doc__', '__file__', '__name__', 'sayhi', 'version']
>>>
>>> dir()
['__builtins__', '__doc__', '__name__', 'mymodule']
```

```
>>> a = 5
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'mymodule']
>>> del a
>>> dir()
['__builtins__', '__doc__', '__name__', 'mymodule']
```

How It Works

First, we see the usage of the `dir` function on the imported `mymodule`. We can see the list of attributes contained in the `mymodule` module.

Next, we use the `dir` function without passing parameters to it, and by default, it returns the list of attributes for the current module. Notice the list of imported modules is also part of this list.

In order to observe the `dir` function in action, we define a new variable `a` by assigning it a value and then we check the output of `dir` and we observe that `a` is now part of the list. We can *undefine* this name `a` by using the `del` statement, and then we observe that `dir` no longer has an `a` in its list.

The `del` statement is used to *delete* a variable/name from the namespace of the current module, and after the `del` statement has been run, you can no longer access that variable/name. It is as if that variable never existed.

Summary

Modules are useful because they help us package services and functionality into reusable pieces. The standard library that comes as part of Python is an example of such a set of modules. We have seen how to use modules and create our own modules.

Next, we will learn about data structures.

Data Structures

Introduction

Data structures are basically just that - they are *structures* which hold some *data* together. In other words, they are used to store a collection of related data.

There are three built-in data structures in Python: list, tuple and dictionary. We will now see how to use each of these data structures.

List

A `list` is a data structure that holds an ordered collection of items - i.e. you can store a *sequence* of items in a list. This is easy to imagine if you think of a shopping list where you have a list of items to buy, except that you probably have each item in a separate line in your shopping list whereas in Python you put commas in between them.

The list of items should be enclosed in square brackets so that Python understands that you are specifying a list. Once you have created a list, you can add, remove or search for items in a list. Since, we can add and remove items, we say that a list is a *mutable* data type - i.e. this type can be altered.

Quick introduction to objects and classes

Although, I've been generally delaying the discussion of objects and classes till now, a little explanation is needed right now so that you can understand lists better. We will still explore this topic in detail in its own chapter later.

A list is an example of usage of objects and classes. When you use a variable `i` and assign a value to it, say integer 5, you can think of it as creating an *object* (instance) `i` of `_class_` (type) `int`. In fact, you can see `help(int)` to understand this better.

A class can have *methods* which are functions designed to be used with that class only. You can use these pieces of functionality only when you have an object of that class. For example, the `list` class has a method called `append` which allows you to add an item to the end of the list. For example, `mylist.append('an item')` will add that string to the list `mylist`. Note the use of dotted notation for accessing methods of the objects, exactly similar to accessing attributes of modules - this uniformity of access and notation is what makes Python remarkably simple.

A class can have *fields* which are variables designed for use with that class only. You can use these variables/names only when you have an object of that class. Fields are also accessed by the dotted notation, for example `mylist.field`.

```
#!/usr/bin/env python
# File name: list1.py

# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print 'I have', len(shoplist), 'items to purchase.'

print 'These items are:', # Notice the comma at the end of the line
for item in shoplist:
    print item,

print ' [end]'
```

```

print 'I also have to buy rice.'
shoplist.append('rice')
print 'My shopping list is now', shoplist

print 'I will sort my list now'
shoplist.sort()
print 'Sorted shopping list is', shoplist

print 'The first item I will buy is', shoplist[0]
olditem = shoplist[0]
del shoplist[0]
print 'I bought the', olditem
print 'My shopping list is now', shoplist

```

Output

```

$ python list1.py
I have 4 items to purchase.
These items are: apple mango carrot banana [end]
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']

```

How It Works

The variable `shoplist` is a shopping list for a person who is going to the market. In `shoplist`, we store the names of the items to buy i.e. only strings. However, a list can hold *any kind of object* including numbers and even other lists.

We have also used the `for...in` loop to iterate over the items of the list. By now, you must have realized that a list is also a sequence. The speciality of sequences will be discussed later in this chapter.

Notice that we use a comma at the end of the `print` statement to suppress the automatic printing of a line break after every `print` statement. This is an ugly way of specifying this, but it is simple and gets the job done.

Next, we add an item to the list using the `append` method of the list object. Then, we check whether the item has been indeed added to the list, by printing the contents of the list by simply passing the list to the `print` statement which prints it in a neat manner.

Then, we sort the list using the `sort` method of the list. This method modifies the list itself and does not return a modified list - this demonstrates the mutable nature of lists, unlike the strings which are immutable.

Next, when we buy the first item from the market, we want to remove it from the list. We do this by using the `del` statement. We mention which item of the list we want to remove and `del` removes it from the list for us. We specify `del shoplist[0]` which removes the first item from the list. Remember that Python starts counting from 0, and not 1.

If you want to find out all the methods defined by the `list` class, see `help(list)`.

Tuple

Tuples are just like lists except that they are *immutable* i.e. you cannot modify tuples. Tuples are defined by specifying items separated by commas within a pair of parentheses. Tuples are usually used in cases where a statement or a function can safely assume that the collection (tuple) of values will not change.

```
#!/usr/bin/env python
# File name: tuple1.py

zoo = ('wolf', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)

new_zoo = ('monkey', 'dolphin', zoo)
print 'Number of animals in the new zoo is', len(new_zoo)
print 'All animals in new zoo are', new_zoo
print 'Animals brought from the old zoo are', new_zoo[2]
print 'Last animal brought over from the old zoo is', new_zoo[2][2]
```

Output

```
$ python tuple1.py
Number of animals in the zoo is 3
Number of animals in the new zoo is 3
All animals in new zoo are ('monkey', 'dolphin', ('wolf', 'elephant',
'penguin'))
Animals brought from the old zoo are ('wolf', 'elephant', 'penguin')
Last animal brought over from the old zoo is penguin
```

How It Works

The variable `zoo` refers to a tuple of items. We see that the `len` function can be used to get the length of the tuple. This also indicates that a tuple is a sequence as well.

We are now shifting these animals to a new zoo since the old zoo is being closed. Therefore, the `new_zoo` tuple contains some animals already present along with the animals brought over from the old zoo. Back to reality, note that a tuple within a tuple does not lose its identity.

We can access the items in the tuple by specifying the item's position within a pair of square brackets, just like we accessed items in a list. This operation is called *subscription* or indexing. We access the third item in `new_zoo` by specifying `new_zoo[2]` and we access the third item in `new_zoo[2]` as `new_zoo[2][2]`. This is simple once you have understood the idiom.

Note for Perl programmers

A list within a list does not lose its identity i.e. lists are not flattened as in Perl. The same applies to a tuple within a tuple, or a tuple within a list, or a list within a tuple, etc. As far as Python is concerned, they are just objects stored using another object, that's all.

Tuple with 0 or 1 items

An empty tuple is constructed by an empty pair of parentheses such as `empty = ()`.

A tuple with a single item is not so simple. You have to specify it with a comma following the first (and only) item so that Python can differentiate between a tuple display and a pair of parentheses surrounding the object in an expression. For example, `singleton = (2,)` indicates a tuple whereas `number = (2)` is just a variable storing a number.

Tuples and the % operator

One of the most common uses of tuples is in association with the % operator, which in turn is used usually with the `print` statement. Here is an example:

```
#!/usr/bin/env python
# File name: tuple2.py

age = 23
name = 'Swaroop'

s = '%s is %d years old' % (name, age)
print s

print 'Why is %s playing with that python?' % name
```

Output

```
$ python tuple2.py
Swaroop is 23 years old
Why is Swaroop playing with that python?
```

How It Works

The % operator is used to substitute parts of the specification string with the corresponding items in the tuple following the % operator. The specification is in the form of %s, %d, etc. where 's' stands for string, 'd' for decimal i.e. integers, etc. The tuple must have items corresponding to these specifications in the same order.

Observe the first usage where we use %s first and this corresponds to the variable `name` which is the first item in the tuple. The second specification is %d corresponding to the integer `age` which is the second item in the tuple. What Python does here is it converts each item in the tuple into a string and substitutes that string value into the place of the specification. Therefore the %s is replaced by the value of the variable `name` and so on.

The % operator makes writing the output very easy and avoids lot of string manipulation to achieve the same. It also avoids using a lot more commas as we have done till now.

Most of the time, you can just use the %s specification and let Python convert the object automatically into a string for you. This works even for numbers. However, using specific %d, etc. is advisable to make sure your program works correctly.

In the second `print` statement, we are using a single specification in the string followed by a single item - there is no tuple. This works only in the case where there is a single specification in the string.

Dictionary

A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name i.e. we associate *keys* (name) with corresponding *values* (details). Note that the key must be unique to have associated values, just like you cannot find the correct information if you have two persons with the exact same name.

Note that you can use only immutable objects (like strings and numbers) for the keys of a dictionary but you can have any kind of objects for the values of the dictionary. This basically translates to say that you should use only simple objects for keys.

Pairs of keys and values are specified in a dictionary by using the notation `d = { key1 :`

value1, key2 : value2 }. Notice that they key-value pairs are separated by a colon and the pairs are separated themselves by commas and all the key-value pairs are enclosed in a pair of curly braces.

The key-value pairs are *not* ordered in any manner. If you want to use them in a particular order, you will have to sort them yourself before using it.

The dictionaries are objects (instances) of the `dict` class.

```
#!/usr/bin/env python
# File name: dict1.py

# 'ab' is short for 'a'ddress'b'ook

ab = {      'Swaroop'   : 'swaroop -at- swaroopch.info',
           'Larry'    : 'larry -at- wall.org',
           'Matz'     : 'matz -at- ruby-lang.org',
           'Spammer'  : 'spammer -at- hotmail.com'
}

print "Swaroop's address is %s" % ab['Swaroop']

# Add a key-value pair
ab['Guido'] = 'guido -at- python.org'

# Deleting a key-value pair
del ab['Spammer']

print '\nThere are %d contacts in the address book\n' % len(ab)

for name, address in ab.items():
    print 'Contact %s at %s' % (name, address)

if 'Guido' in ab:          # OR ab.has_key('Guido')
    print "\nGuido's address is %s" % ab['Guido']
```

Output

```
$ python dict1.py
Swaroop's address is swaroop -at- swaroopch.info

There are 4 contacts in the address book

Contact Swaroop at swaroop -at- swaroopch.info
Contact Larry at larry -at- wall.org
Contact Matz at matz -at- ruby-lang.org
Contact Guido at guido -at- python.org

Guido's address is guido -at- python.org
```

How It Works

We create the dictionary `ab` using the syntax already discussed. Then, we access the key-value pairs by specifying the key using the indexing/subscript operator. Observe that the syntax is simple for dictionaries as well.

We can add new key-value pairs by simply using the indexing operator to access a key and assigning a value, as we have done for Guido in the above program.

We can delete key-value pairs using the `del` statement. We simply pass the dictionary using the indexing operator and the key to the `del` statement which promptly deletes the key and the

corresponding value from the object.

Next, we access each key-value pair of the dictionary using the `items` method of the dictionary which returns a list of tuples where each tuple contains a pair of items - the key followed by the value. We retrieve this pair and assign it to the variables `name` and `address` correspondingly for each pair using the `for . . in` loop and then print these values within the for-block.

We can check if a key-value pair exists using the `in` operator or the `has_key` method of the `dict` class. You can see the documentation for the complete list of methods of the `dict` class using `help(dict)`.

Keyword arguments and dictionaries

If you have used keyword arguments in functions, you have already used dictionaries! Just think about it - the key-value pairs are specified by you in the parameter list of the function definition and when you access parameters within the function, it is just a key access of a dictionary.

This dictionary represents the *namespace* of that object, whether it is a function, module or an instance/object of a class. This namespace dictionary is also referred to as a *symbol table* in compiler design terminology.

Sequences

Now, I can finally describe what sequences are. Lists, tuples and strings are examples of sequences, but what are sequences and what is so special about them? Two main features of a sequence is the *indexing* operation and the *slicing* operation which allows us to retrieve a slice of the sequence i.e. a part of the sequence.

```
#!/usr/bin/env python
# File name: seq.py

shoplist = ['apple', 'mango', 'carrot', 'banana']

# Indexing or 'subscription' operation
print 'Item 0 is', shoplist[0]
print 'Item 1 is', shoplist[1]
print 'Item 2 is', shoplist[2]
print 'Item 3 is', shoplist[3]
print 'Item -1 is', shoplist[-1]
print 'Item -2 is', shoplist[-2]

# Slicing of a list
print 'Item from position 1 up to position 3 is', shoplist[1:3]
print 'Item from position 2 up to end is', shoplist[2:]
print 'Item from position 1 up to position -1 is', shoplist[1:-1]
print 'Item from start up to end is', shoplist[:]

# Slicing on a string
name = 'swaroop'
print 'Characters from position 1 up to position 3 is', name[1:3]
print 'Characters from position 2 up to end is', name[2:]
print 'Characters from position 1 up to position -1 is', name[1:-1]
print 'Characters from start up to end is', name[:]
```

Output

```
$ python seq.py
Item 0 is apple
Item 1 is mango
Item 2 is carrot
```

```

Item 3 is banana
Item -1 is banana
Item -2 is carrot
Item from position 1 up to position 3 is ['mango', 'carrot']
Item from position 2 up to end is ['carrot', 'banana']
Item from position 1 up to position -1 is ['mango', 'carrot']
Item from start up to end is ['apple', 'mango', 'carrot', 'banana']
Characters from position 1 up to position 3 is wa
Characters from position 2 up to end is aroop
Characters from position 1 up to position -1 is waroo
Characters from start up to end is swaroop

```

How It Works

First, we see how to use indexes to get individual items of a sequence. This is also referred to as the *subscription* operation. Whenever you specify a number to a sequence within square brackets, Python fetches the item located in the numbered location in the sequence. Remember that Python starts counting from 0. hence, `shoplist[0]` fetches the first item and `shoplist[3]` fetches the fourth item in the `shoplist` sequence.

The index can also be a negative number, in which case the position is calculated from the end of the sequence. Therefore, `-1` refers to the *last* position in the sequence, `-2` refers to the *last but one* position in the sequence, and so on.

The slicing operation is used by specifying the name of the sequence followed by an optional pair of numbers separated by a colon within square brackets. Note that this is very similar to the indexing operation. The numbers are optional for slicing but the colon has to be present.

The first number before the colon in the slicing operation refers to the position from where the slice starts and the second number after the colon indicates where the slice will stop at. If the first number is not specified, Python will start at the beginning of the original sequence. If the second number is not specified, Python will stop at the end of the original sequence. Note that the slice returned `_starts_` at the start position and will end just before the *end* position i.e. the start position is included in the slice but the end position is excluded from the slice.

Thus, `shoplist[1:3]` returns a new sequence which is a slicing of the original sequence starting at position 1, includes position 2 but stops at position 3, and therefore this slice contains only two items. Similarly, `shoplist[:]` returns a copy of the whole sequence.

You can also do slicing with negative positions. Negative numbers are counted from the end of the sequence. For example, `shoplist[:-1]` will return a slice of the sequence which excludes the last item of the sequence.

For example, if you want to remove ending newlines in a string, we can do this:

```

>>> s = 'This is a line.\n'
>>> s
'This is a line.\n'
>>> if s.endswith('\n'):      # 'endswith' is a method of the str class
...     s = s[:-1]
...
>>> s
'This is a line.'

```

Try various combinations of such slice specifications using the interactive interpreter prompt and you can see the results immediately. The great thing about sequences is that you can access tuples, lists and strings all in the same way!

References

When you create an object and assign it to a variable, the variable only *refers* to the object and does not represent the object itself! In other words, the variable name just points to that part of your computer's memory where the object is stored. This is called as *binding* of the name to the object.

Generally, you do not need to be worried about this, but there is a subtle effect due to references which you need to be aware of. This is demonstrated by the following example.

```
#!/usr/bin/env python
# File name: ref.py

print 'Simple assignment'
shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist # mylist is just another name pointing to the same object
print 'shoplist is', shoplist
print 'mylist is', mylist

print 'Removing first item'
del shoplist[0] # I purchased the first item, so I am removing it from
the list

print 'shoplist is', shoplist
print 'mylist is', mylist
# Notice that both shoplist and mylist both print the same list

print 'Copy by making a full slice'
mylist = shoplist[:] # Make a copy by doing a full slicing
del mylist[0] # Remove first item

print 'shoplist is', shoplist
print 'mylist is', mylist
# Notice that now shoplist and mylist are different
```

Output

```
$ python ref.py
Simple assignment
shoplist is ['apple', 'mango', 'carrot', 'banana']
mylist is ['apple', 'mango', 'carrot', 'banana']
Removing first item
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

How It Works

Notice that when the statement `mylist = shoplist` runs, both of these names point to the same object, so when we remove the first item in `shoplist`, `mylist` also shows the item to be removed.

If we want to make a real copy of `shoplist` and store it as `mylist`, then we make use of slicing since this is a sequence and we generate a full copy.

The point to understand is that assigning one name to another name only results in both of the names using the same reference to an object, and it does *not* cause a copy of the object to be stored.

Note for Perl programmers

Remember that an assignment statement for lists does *not* create a copy. You have to use slicing operation to make a copy of the sequence.

Summary

We have explored various built-in data structures of Python. These data structures will be essential for writing programs of reasonable size.

I highly recommend reading `help(str)`, `help(list)`, `help(tuple)`, `help(dict)` before proceeding further.

Next, we will see how to design and write a real-world Python program.

Problem Solving

Introduction

We have explored various parts of Python language, so let us now take a look at a simple example of how these many parts fit together, by designing and writing a program which *does* something useful. Note that we still have many parts of Python to explore, but let us have some fun writing a program before we proceed further.

The problem

The problem I want to solve is *I want a program which creates a backup of all my important files.*

Although, this is a simple problem, there is not enough information for us to get started with the solution. A little more **analysis** is required. For example, how do we specify which files are to be backed up? Where is the backup stored? How are they stored in the backup?

After analyzing the problem properly, we **design** our program. We make a list of things about how our program should work. In this case, I have created the following list on how *I* want it to work. If you do the design, you may not come up with the same kind of design - every person has their own way of doing things, that is okay.

1. The files and directories to be backed up are specified in a list.
2. The backup must be stored in a main backup directory.
3. The files are backed up into a zip file.
4. The name of the zip archive is the current date and time
5. We use the standard `zip` command available by default in any Linux/BSD/Mac OS X system. Windows users can use the `Info-Zip` program for the same purpose. Note that you can use any archiving command you want to use as long as it has a command line interface so that we can pass arguments to it from our script.

The Solution

As the design of our program is now stable, we can write the code which is an **implementation** of our solution.

The First Version

```
#!/usr/bin/env python
# File name: backup1.py

import os, time

# 1. The files and directories to be backed up are specified in a list.
source = ['/Users/swaroopch/Documents', '/Users/swaroopch/Code']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or
something like that

# 2. The backup must be stored in a main backup directory.
target_directory = '/Users/swaroopch/Backup/'

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_directory + time.strftime('%Y%m%d_%H%M%S') + '.zip'
```

```
# 5. We use the standard ``zip`` command to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))
print zip_command

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

Output

```
$ python backup1.py
zip -qr
'/Users/swaroopch/Backup/20051113_234436.zip' /Users/swaroopch/Documents/ /Users
/swaroopch/Code/
Successful backup to /Users/swaroopch/Backup/20051113_234436.zip
```

Now, we are in the **testing** phase where we test that our program works properly. If it doesn't behave as expected, then we have to **debug** our program i.e. remove the *bugs* (errors) from the program.

How It Works

You will notice that we have converted our *design* into *code* in a step-by-step manner.

We make use of the `os` and `time` modules and so we import them into our namespace. Then, we specify the files and directories to be backed up in the `source` list. The target directory is where we store all the backup files and this is specified in the `target_directory` variable. The name of the zip archive that we are going to create is the current date and time which we fetch using the `time.strftime` function. It will also have the `.zip` extension and will be stored in the `target_directory` directory.

The `time.strftime` function takes a specification such as the one we have used in the above program. The `%Y` specification will be replaced by the year without the century. The `%m` specification will be replaced by the month as a decimal number between 01 and 12 and so on. The complete list of such specifications can be found in the official Python documentation that comes with your Python distribution. Notice that these specifications are similar to the specifications used for the `%` operator.

We create the name of the target zip file using the addition operator which *concatenates* the strings i.e. it joins the two strings together and returns a new one. Then, we create a string `zip_command` which contains the command that we are going to execute. You can check whether this command works by running it yourself on a shell prompt.

The `zip` command that we are using has some options and parameters passed. The `-q` option is used to indicate that the zip command should work **quietly**. The `-r` option specifies that the zip command should work **recursively** for directories i.e. it should include subdirectories and the files within the subdirectories also. The two options are combined and specified in a shorter fashion as `-qr`. The options are followed by the name of the zip archive to be created, followed by the list of the files and directories to backup. We convert the `source` list into a string using the `join` method of strings which concatenates each item of the list into a single string with the original string (whose `join` method is called) to be interleaved in-between each item. In our case, the result is a space-separated list of file names and directories.

Then, we finally *run* the command using the `os.system` function which runs the command as if it was run from the `_system_` i.e. in the shell - it returns 0 if the command was successful, else it

returns an error number.

Depending on the outcome of the command, we print the appropriate message that the backup has succeeded or failed.

That's it, we have created a script to take a backup of our important files!

Note to Windows users

You can set the `source` list and `target` directory to any file and directory names but you have to be a little careful in Windows. The problem is that Windows uses the backslash (`\`) as the directory separator character but Python uses backslashes to represent escape sequences. So, you have to represent a backslash by using the `\\` escape sequence or you have to use raw strings. For example, use `'C:\\Documents'` or `r'C:\Documents'` but do **not** use `'C:\Documents'` where you are using an unknown escape sequence `\D`.

Now that have a working backup script, we can use it whenever we want to take a backup of the files. Linux/BSD/Mac OS X users are advised to use the executable python programs method (as discussed earlier) so that they can run the backup script from anywhere any time. This is called the **operation** phase or **deployment** phase.

The above program works properly, but usually first programs do not work exactly as you expect. For example, there might be problems if you have not designed the program properly or you might have made mistakes in typing the code, etc. Appropriately, you will have to go back to the design phase or you will have to debug your program.

The Second Version

The first version of our script works. However, we can make some refinements to the program so that it can work better. This is called the **maintenance** phase of the software.

One of the refinements I felt was useful was a better file-naming mechanism - using the *time* as the name of the file within a directory with the current date as a directory within the main backup directory. One advantage is that your backups are stored in a hierarchical manner and is easier to manage. Another advantage is that the length of the file names are much shorter this way. Yet another advantage is that separate directories will help you to easily check if you have taken a backup for each day since the directory will be created only if you have taken a backup for that day.

```
#!/usr/bin/env python
# File name: backup2.py

import os, time

# 1. The files and directories to be backed up are specified in a list.
source = ['/Users/swaroopch/Documents', '/Users/swaroopch/Code']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or
something like that

# 2. The backup must be stored in a main backup directory.
target_directory = '/Users/swaroopch/Backup/'

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
today = target_directory + time.strftime('%Y%m%d')
# The current time is the name of the zip archive
now = time.strftime('%H%M%S')

# Create the subdirectory if it does not exist already
if not os.path.exists(today):
```

```

    os.mkdir(today)
    print 'Successfully created directory', today

# The name of the zip file
target = os.path.join(today, now + '.zip')

# 5. We use the standard ``zip`` command to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))
print zip_command

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'

```

Output

```

$ python backup2.py
Successfully created directory /Users/swaroopch/Backup/20051114
zip -qr
'/Users/swaroopch/Backup/20051114/001955.zip' /Users/swaroopch/Documents/ /Users
/swaroopch/Code/
Successful backup to /Users/swaroopch/Backup/20051114/001955.zip

```

How It Works

Most of the program remains the same. The changes is that we check if there is a directory with the current day as name, inside the main backup directory using the `os.exists` function. If it doesn't exist, we create it using the `os.mkdir` function.

Notice the use of the `os.path.join` function that converts a list of names (directories one below the other and finally the file (optional)) into a full path. Using `os.path.join` makes our program portable across different systems.

The Third Version

The second version works fine when I do many backups, but when there are lots of backups, I am finding it hard to differentiate what the backups were for! For example, I might have made some major changes to a program or presentation, then I want to associate what those changes are with the name of the zip archive. This can be easily achieved by attaching a user-supplied comment to the name of the zip archive.

```

#!/usr/bin/env python
# File name: backup3.py

import os, time

# 1. The files and directories to be backed up are specified in a list.
source = ['/Users/swaroopch/Documents', '/Users/swaroopch/Code']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or
something like that

# 2. The backup must be stored in a main backup directory.
target_directory = '/Users/swaroopch/Backup/'

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
today = target_directory + time.strftime('%Y%m%d')
# The current time is the name of the zip archive

```

```

now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip archive
comment = raw_input('Enter a comment --> ')
if len(comment) == 0:          # check if a comment was entered
    target = os.path.join(today, now + '.zip')
else:
    target = os.path.join(today, now + '_' * comment.replace(' ', '_') + '.zip')

# Create the subdirectory if it does not exist already
if not os.path.exists(today):
    os.mkdir(today)
    print 'Successfully created directory', today

# 5. We use the standard ``zip`` command to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))
print zip_command

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'

```

Output

```

$ python backup3.py
Enter a comment --> Added new chapter
Traceback (most recent call last):
  File "programs/backup3.py", line 25, in ?
    target = os.path.join(today, now + '_' * comment.replace(' ', '_') + '.zip')
TypeError: can't multiply sequence by non-int

```

How It (Does Not) Work

This program does not work!

We carefully observe the error message printed by Python and we check why is there a multiplication being done here? Looks like there has been a mistake, it should have been + instead of the * that we have typed there in line 25. So, we correct that mistake and try running the program again. This is called **bug fixing**.

The Fourth Version

```

#!/usr/bin/env python
# File name: backup4.py

import os, time

# 1. The files and directories to be backed up are specified in a list.
source = ['/Users/swaroopch/Documents', '/Users/swaroopch/Code']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or
something like that

# 2. The backup must be stored in a main backup directory.
target_directory = '/Users/swaroopch/Backup/'

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
today = target_directory + time.strftime('%Y%m%d')
# The current time is the name of the zip archive

```

```

now = time.strftime('%H%M%S')

# Take a comment from the user to create the name of the zip archive
comment = raw_input('Enter a comment --> ')
if len(comment) == 0:          # check if a comment was entered
    target = os.path.join(today, now + '.zip')
else:
    target = os.path.join(today, now + '_' + comment.replace(' ', '_') + '.zip')
# Bug fixed!

# Create the subdirectory if it does not exist already
if not os.path.exists(today):
    os.mkdir(today)
    print 'Successfully created directory', today

# 5. We use the standard ``zip`` command to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))
print zip_command

# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'

```

Output

```

$ python backup4.py
Enter a comment --> added new chapter
zip -qr
'/Users/swaroopch/Backup/20051114/004004_added_new_chapter.zip' /Users/swaroopch
/Documents/ /Users/swaroopch/Code/
Successful backup
to /Users/swaroopch/Backup/20051114/004004_added_new_chapter.zip

$ python backup4.py
Enter a comment -->
zip -qr
'/Users/swaroopch/Backup/20051114/004235.zip' /Users/swaroopch/Documents/ /Users
/swaroopch/Code/
Successful backup to /Users/swaroopch/Backup/20051114/004235.zip

```

How It Works

This program now works! We have corrected the mistakenly-entered * and replaced it with the actually-required +.

Let us now discuss the actual enhancements made in version 3. We take the user's comments using the `raw_input` function and then check if the user actually entered something by finding out the length of the input using the `len` function. If the user has just pressed the enter key without entering a comment (maybe it was a routine backup or no noteworthy changes were made), then we proceed as we have done before.

However, if a comment was supplied, then this comment is attached to the name of the zip archive just before the `.zip` extension. Notice that we are replacing spaces in the comments with underscores - this is because managing file names with underscores are much easier.

More refinements

The fourth version is a satisfactorily working script for most users, but there is always room for

improvement. For example, you can include a *verbosity* level for the program where you can specify a `-v` option to make your program become more talkative.

Another possible enhancement would be to allow extra files and directories to be passed to the script at the command line. We will get these from the `sys.argv` list and then we add them to our source list using the `list.extend` method.

Yet another refinement would be to use the `tar` command instead of the `zip` command. One advantage is that when you use the `tar` command along with the `gzip` command, the backup is much faster and the backup archive created is also much smaller. If I need to use this archive in Windows, then WinZip can handle such archive in Windows and other platforms have built-in commands to handle `.tar.gz` files. Windows users can [download](#) and install the `tar` command as well.

The command string will now be:

```
tar = 'tar -cvzf %s %s -X /Users/swaroopch/Documents/excludes.txt'  
      % (target, ' '.join(srcdir))
```

The options used for the `tar` command are explained below.

- `-c` indicates Creation of an archive.
- `-v` indicates Verbose i.e. the command should be more talkative.
- `-z` indicates that the gZip filter should be used.
- `-f` indicates Force in creation of archive i.e. it should replace if there is a file by the same name already.
- `-X` indicates a file which contains a list of filenames which must be eXcluded from the backup. For example, you can specify that `*.pyc` in this file to *not* include any filenames with that pattern in the backup.

Important

The most preferred way of creating such kind of archives would be using the `zipfile` or `tarfile` module respectively, available as part of the standard library. Using these libraries avoids usage of the `os.system` which is generally advisable not to use because it is very easy to make dangerous operations/costly mistakes using it.

However, I have been using the `os.system` way of creating a backup, for pedagogical purposes, so that the example is simple enough to be understood by everybody but real enough to be useful and interesting.

Phases

We have now gone through various **phases** in the process of writing a software. These phases can be summarised as follows:

1. What (Analysis)
2. How (Design)
3. Do It (Implementation)
4. Test (Testing and Debugging)
5. Maintain (Refinement)

A recommended way of writing programs is the procedure we have followed in creating the backup script - Do the analysis and design. Start implementing with a simple version. Test and debug it thoroughly. Use it to ensure that it works as expected. Now, add any features that you want, immediately continue the Do It-Test-Use cycle as many times as required.

Remember that *Software is grown, not built*.

Summary

We have seen how to create our own Python programs and the various phases involved in writing such programs. You may find it useful to create your own program just like the one in this chapter so that you become comfortable with Python as well as problem-solving.

Next, we will discuss object-oriented programming.

Object oriented programming

Introduction

In the programs that we have discussed till now, we have designed our program around functions and blocks of statements, which manipulate data. This is called the *procedure-oriented* way of programming. There is another way of designing and organizing your program which is to combine data and functionality and wrap it inside what is called an *object*. This is called *object-oriented* programming. Most of the time you can use procedural programming but sometimes when you want to write large programs or have a solution better suited to it, you can use object-oriented programming techniques.

Classes and objects are the two main aspects of object-oriented programming. A **class** creates a new *type* where **objects** are *instances* of the class. An analogy is that you have variables of type `int` which means that variables that store integers are variables which are instances/objects of the `int` class.

Note for C++/Java/C# Programmers

Note that even integers are treated as objects (see `help(int)`). This is unlike C++ or Java (< 1.5) where integers are native "magic" types.

C# and Java (>= 1.5) programmers will recognize that this concept is similar to the *boxing and unboxing* concept.

Objects can store data using ordinary variables that *belong* to the object. Variables that belong to an object or class are called **fields**. Objects can also have functionality by using functions that *belong* to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which can be used independently but fields and methods have to be used with a class or object. Collectively, the fields and methods can be referred to as the **attributes** of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called **instance variables** and **class variables** respectively.

A class is created using the `class` keyword. The fields and methods of the class are listed in a block.

The self

Class methods have only one specific difference from ordinary functions - they must have an extra parameter added to the beginning of the parameter list, but you *do not* supply a value for this parameter when you call the method because Python will supply the argument. This particular parameter refers to the object itself whose method is being called, and by convention it is called `self`.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and your editor can highlight it in a special color as well.

Note for C++/Java/C# Programmers

The `self` is equivalent to the `this` pointer in C++ and the `this` reference in Java and C#.

You must be wondering how Python gives the value for `self` and why you do not need to give a value for it, even though you have to explicitly specify it in the method parameter list. An example will make this clear. Let us say you have a class called `MyClass` and an instance of this class is called `MyObject`. When you call a method of this object as `MyObject.method(arg1, arg2)`, this is automatically converted into `MyClass.method(MyObject, arg1, arg2)` - this is what the special `self` is all about.

This also means that if you have a method which takes no arguments, then you still have to define the method to take a simple parameter - the `self`.

Classes

The simplest class possible is demonstrated in the following example.

```
#!/usr/bin/env python
# File name: class1.py

class Person:
    pass

p = Person()
print p
```

Output

```
$ python class1.py
<__main__.Person instance at 0x6c2b0>
```

How It Works

We create a new class using the `class` statement followed by the name of the class. This is followed by an optional list of classes enclosed in parentheses. Here, we do not specify this - it will be explained in detail later in this chapter. Then, the line ends with a colon, and the fields and methods of the class have to be specified in a block of statements. However, we do not declare any fields or methods here and we indicate an empty block using the `pass` statement.

Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses. We will learn more about instantiation later in this chapter. For our verification, we confirm the type of the variable by simply printing it. It tells us that we have an instance of the `Person` class in the `__main__` module.

Notice how instantiating an instance of a class is remarkably similar to calling a function. This is yet another example of simplicity advocated by Python. Why is this remarkable? Because you can use the class or function without knowing how it works underneath. For example, remember that we earlier used `int` to convert a string to an integer? Well, it was actually a class, and we used `int('5')` to get the integer 5! See `help(int)` for details.

Notice that the address of the computer memory where your object is stored is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

Object Methods

We have already discussed that classes and objects can have methods which are functions with the extra `self` parameter. We will now see an example.

```
#!/usr/bin/env python
# File name: method.py

class Person:
    def say_hi(self):
        print 'Hello, how are you?'

p = Person()
p.say_hi()

# This short example can also be written as Person().say_hi()
```

Output

```
$ python method.py
Hello, how are you?
```

How It Works

Here we see the `self` in action. Notice that the `say_hi` method takes no parameters but still has the `self` in the parameter list of the function definition.

The `__init__` method

There are many method-names which have special significance in Python classes. We will see the significance of the `__init__` method in this section.

The `__init__` method is run as soon as an object of this class is instantiated. The method is useful to do any `_initialization_` you want to do with your object. Notice the double underscore both in the beginning and at the end of the name.

```
#!/usr/bin/env python
# File name: class_init.py

class Person:

    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print 'Hello, my name is', self.name

p = Person('Swaroop')
p.say_hi()
```

Output

```
$ python class_init.py
Hello, my name is Swaroop
```

How It Works

We define the `__init__` method as taking a parameter `name` (after the `self`), and then assign the value of the `name` parameter and store it as the field also called `self.name`. Notice that these are two different variables even though they have the same name. This is because they have different *namespaces*. The `name` lives within the `__init__` method and the `self.name` lives within that particular instance. The dotted notation helps us to differentiate between them.

Now, we are able to use the `self.name` field in the `say_hi` method and use the `self.name` to print a more appropriate and specific message.

Note for C++/Java/C# Programmers

The `__init__` method is analogous to a `_constructor_` method in C++, C# and Java.

Class and Object Variables

We have already discussed the functionality part of classes and objects, now we will see the data part of it. There is nothing fancy about fields except that they are names which are *bound* to the class or object **namespaces** i.e. the names exist within the class or object respectively only. Class variables are variables *owned* by the class. Object variables are variables *owned* by the object.

Class variables are shared in the sense that they are accessible by all objects/instances of that class. There is only one copy of the class variable and when any one object makes a change to the class variable, the change is reflected in all other instances as well.

Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance of the same class. An example will make this easy to understand.

```
#!/usr/bin/env python
# File name: objvar.py

class Person:
    '''Represents a person.'''
    population = 0                # class variable

    def __init__(self, name):
        '''Initializes the person's data.'''
        self.name = name         # object variable
        print '(Initializing %s)' % self.name

        # When this person is created, he/she adds to the population
        Person.population += 1

    def __del__(self):
        '''I am dying.'''
        print '%s says bye.' % self.name

        Person.population -= 1

        if Person.population == 0:
            print 'I am the last one.'
        else:
            print 'There are still %d people left.' % Person.population

    def say_hi(self):
        '''Greeting by the person.

        Really, that's all it does.'''
        print 'Hi, my name is %s.' % self.name

    def how_many(cls):
        '''Prints the current population.'''
        if Person.population == 0:
            print 'Nobody is alive as of now.'
        elif Person.population == 1:
```

```
        print 'There is just one person here.'
    else:
        print 'We have %d persons here.' % Person.population
how_many = classmethod(how_many)
```

```
swaroop = Person('Swaroop')
swaroop.say_hi()
Person.how_many()
```

```
kalam = Person('Abdul Kalam')
kalam.say_hi()
Person.how_many()
```

```
swaroop.say_hi()
Person.how_many()
```

Output

```
$ python objvar.py
(Initializing Swaroop)
Hi, my name is Swaroop.
There is just one person here.
(Initializing Abdul Kalam)
Hi, my name is Abdul Kalam.
We have 2 persons here.
Hi, my name is Swaroop.
We have 2 persons here.
Abdul Kalam says bye.
There are still 1 people left.
Swaroop says bye.
I am the last one.
```

How It Works

This is quite a long example but helps us to demonstrate many aspects about classes and objects.

The `population` variable belongs to the `Person` class and hence is a class variable. The `name` variable belongs to the object since it is assigned using `self`, and hence is an object variable. Notice that we refer to the `population` class variable as `Person.population` and not as `self.population`. Remember that the class variable belongs to the class and should be accessed by dotted notation using the class name and the object variable belongs to the object and should be dotted notation using the object which is `self`.

The `__init__` method is used to initialize the `Person` instance. Here, we initialize the class by providing a name so that each instance can have a name. We initialize an object variable `self.name` with the same value as `name`. We also increment the `Person.population` by 1 because a new person has been added to the population.

Within a method (including `__init__` and other special methods as well), you can refer to the methods and fields that belong to that object using the `self.method` or `self.field` dotted notation. This is called an `_attribute reference_`.

We also see the use of *docstrings* for classes as well as for methods. We can access the docstring of the class at runtime using the `Person.__doc__` field and the method docstring as `Person.say_hi.__doc__`.

On similar lines as `__init__`, the `__del__` method is a special method that is called when an object is going to die i.e. the object is no longer being used and is being destroyed by the system to

free up memory. For our purposes, we just decrement the population by 1. Remember that when we say the `__del__` method is special, it means that it has special semantics because it is called by Python automatically when the object is going to be destroyed, there is no special meaning on how the method itself works or is defined.

It's important to note that all data members i.e. fields are *public* by default, they can be accessed by any other object. You can use attributes that have a double underscore prefix which will not be accessible by other objects. For example, if you have `self.__private`, then that field will not be accessible by other objects.

Note for C++/Java/C# Programmers

All methods are *virtual* in Python.

The convention followed by most Python projects is that any variable that is supposed to be "internal" to how the class/object works should begin with an underscore (such as `self.__private`) and all other names are public and can be used by other classes/objects. Remember that this is only a convention and not enforced by Python.

Note for C++/Java/C# Programmers

The `__del__` method is analogous to the concept of a *destructor*.

Inheritance

One of the major benefits of object-oriented programming is **code reuse** and one of the methods to achieve this code reuse is through the *inheritance* mechanism. Inheritance can be best imagined as implementing a *type and subtype* relationship between classes.

Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific characteristics such as salary, courses and leaves for teachers, and marks and fees for students.

You can create two independent classes for each type and process them but adding a new common characteristic would mean adding to both of these independent classes. This quickly becomes problematic.

A better alternative would be to create a common class called `SchoolMember` and then have the teacher and student classes *inherit* from this class i.e. they will become sub-types of this type and then we can add the specific characteristics to these sub-types.

There are many advantages to this approach. If we add/change any functionality in `SchoolMember`, this is automatically reflected in the sub-types as well. For example, you can add a new ID-card field for both teachers and students by simply adding it to the `SchoolMember` class. However, changes in the sub-types do not affect the parent class or other sub-types.

Another advantage is that you can collectively process both teachers and students by treating them as `SchoolMembers`. This can be useful in situations such as counting of the number of school members. This is called as **polymorphism** where a sub-type can be substituted in any situation where a parent type is expected.

Observe that the main objective of *code reuse* is achieved because we have common characteristics only in the parent class which is re-used by the sub-classes.

The `SchoolMember` class can be called a *base class* or the *superclass*. The `Teacher` and `Student` classes can be called as *derived classes* or *subclasses*.

Let us now write down this example and try it out.

```
#!/usr/bin/env python
# File name: inherit.py

class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember %s)' % self.name

    def __str__(self):
        '''Represent the school member as a string.'''
        return 'Name:"%s" Age:"%s"' % (self.name, self.age)

class Teacher(SchoolMember):
    '''Represents a teacher.'''

    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher %s)' % self.name

    def __str__(self):
        return '%s Salary:"%d"' % (SchoolMember.__str__(self), self.salary)

class Student(SchoolMember):
    '''Represents a student.'''

    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student %s)' % self.name

    def __str__(self):
        return '%s Marks:"%d"' % (SchoolMember.__str__(self), self.marks)

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 23, 75)

print          # prints a blank line

members = [t, s]
for member in members:
    print member          # works for both Teachers and Students
```

Output

```
$ python inherit.py
(Initialized SchoolMember Mrs. Shrividya)
(Initialized Teacher Mrs. Shrividya)
(Initialized SchoolMember Swaroop)
(Initialized Student Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary:"30000"
Name:"Swaroop" Age:"23" Marks:"75"
```

How It Works

To use inheritance for a class, we specify the base class names in a tuple after the class name in the class definition.

Notice that in the derived classes, we explicitly call the `__init__` method of the base class so that the superclass portion of the object can be initialized. This is very important to remember - Python does not automatically call the constructor of the base class, you have to explicitly call it yourself.

Also observe, that we can call methods of the base class using dotted notation and explicitly passing `self` to those methods.

The `__str__` is another method with special semantics - this method is called when Python wants to `_stringify_` the object. For example, in our program we say `print member` and we need a way of printing that object, so Python calls the `__str__` method of that object and promptly prints the output of that method to the screen.

Notice that we can treat instances of `Teacher` or `Student` as instances of `SchoolMember` when we use the `for . . in` loop and get the name and age of the `SchoolMember`. This is where polymorphism comes into the picture.

If the `Teacher` or `Student` class did not have a method such as its own `__str__`, then Python would start looking for a method of the same name in its list of base classes, which in this case would be `SchoolMember` which does have a `__str__` defined, and it will then call that method. You can try this out by removing the `__str__` definition in any of the derived classes.

Summary

We have now explored many aspects of classes and objects as well as the various terminologies associated with it. We have also seen the benefits and pitfalls of object-oriented programming.

Python is a highly object-oriented programming language, so understanding the nuances of objects and classes will help a lot in writing and understanding how Python programs work.

Next, we will learn how to deal with input/output and how to access files in Python.

Input Output

Introduction

There will be plenty of problems where you will want your program to interact with the user (which could be yourself). For example, you might want to take some input from the user, perform some calculations and then print some results. We have seen how to do this using `raw_input` and `print` respectively. We can also use the various methods of the `str` class, such as the `rjust` method to get a string which is right justified to a specified width. See `help(str)` for details.

Another common type of input/output is dealing with files. The ability to create, read and write files is essential to many programs and we will explore this aspect in this chapter.

Files

You can open and use files for reading or writing, by creating an instance of the `file` class and using its `read` or `write` methods respectively. Finally, when you are done with using the file, you have to `close` the file object so that Python can flush the buffers and make sure everything is written to the disk.

```
#!/usr/bin/env python
# File name: using_file.py

poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
'''

f = file('poem.txt','w')          # open for 'w'riting
f.write(poem)                    # write text to file
f.close()                        # close the file

f = file('poem.txt')             # if no mode is specified, 'r'ead mode is
assumed by default
while True:
    line = f.readline()
    if len(line) == 0:            # zero length indicates end-of-file
        break
    print line,                  # notice comma to avoid automatic line breaks
f.close()                        # close the file
```

Output

```
$ python using_file.py
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!

$ cat poem.txt          # 'cat' prints the file to the screen
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```



```
f = file(shoplist_file)
storedlist = pickle.load(f)           # convert back to a Python object
print storedlist
```

Output

```
$ python pickling.py
['apple', 'mango', 'carrot']
```

```
$ cat shoplist.data
(lp1
S'apple'
p2
aS'mango'
p3
aS'carrot'
p4
a.
```

How It Works

Notice that we use the `import...as` syntax to give alternate names to imported modules. So, by changing a single statement, we can use alternative modules that have the same list of methods and fields that we use. In this case, `cPickle` and `pickle` are modules that have the same interface but different mechanisms underneath.

To store an object in a file, we need to first create a file object in write mode and then we can store the object in the file by calling the `dump` function of the `pickle` module. This process is called **pickling** in Python, and is also referred to as *serializing*.

We can retrieve the object using the `load` function of the `pickle` module which then converts the string representation of the object back to a Python object. This process is called **unpickling**, and is also referred to as *unserializing*.

More about Strings

Using Triple Quotes

You can specify multi-line strings using triple quotes (`'''` or `"""`). You can use single quotes and double quotes freely within the triple quotes.

An example is

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

Escape Sequences

Suppose you want to have a string which contains a single quote (`'`), how will you specify this string? For example the text is `What's your name?` You cannot specify `'What's your name?'` because Python will get confused as to where the string starts and ends. So, you will have to specify that this single quote in `What's` does not indicate the end of the string. This can be done

with the help of an *escape sequence*. You can specify the escape sequence for the single quote as `\'` - notice the backslash. Now, you can specify the string as `'What\'s your name'`.

Another way of specifying this specific string would be `"What's your name?"` i.e. using double quotes. Similarly, you have an escape sequence for using a double quote itself in a double quoted string. Also, you can use a backslash itself using the escape sequence `\\`.

What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown above or you can use a newline escape sequence `\n` to indicate the start of a new line. An example is `This is the first line.\nThis is the second line`. Another useful escape sequence is the tab `\t`. There are many more escape sequences but we have discussed only the most commonly used ones here.

One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added.

For example,

```
"This is the first sentence.\nThis is the second sentence."
```

is equivalent to

```
"This is the first sentence.This is the second sentence."
```

Raw Strings

If you need to specify some strings where no special processing such as escape sequences are to be handled, then what you need is to specify a `_raw_` string by prefixing `r` or `R` to the string. An example is `r"Newlines are indicated by \n"`.

Note for Regular Expression users

Always use raw strings when dealing with regular expressions. Otherwise, a lot of backwhacking may be required. For example, backreferences can be referred to as `'\\1'` or `r'\1'`.

Unicode Strings

Unicode is a standard way of writing international text. If you want to write text in your native language such as Hindi or Arabic, then you need to have a Unicode-enabled text editor. Similarly, Python allows you to handle Unicode text - all you need to do is prefix `u` or `U` to the string specification. For example, `u"This is a Unicode string."` is a Unicode string.

Remember to use Unicode strings when you are dealing with text files, especially when you know that the file will contain text written in languages other than English.

Strings are immutable

This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really is not. We will see why this is not a limitation in the various programs that we will see later on.

String literal concatenation

If you place two string literals side by side, they are automatically concatenated by Python. For

example, 'What\'s your name?' is automatically converted to "What's your name?".

Summary

We have discussed various types of input/output, file handling and using the pickle module.

Next, we will explore the concept of exceptions.

Exceptions

Introduction

Exceptions occur when *exceptional* situations occur in your program. For example, what if you are going to read a file and the file does not exist? What if you accidentally deleted the file when your program was running? Such situations can be handled using **exceptions**.

Another scenario is what if your program had invalid statements? Python *raises* its hands and tells you there is an *error*.

Errors

Consider a simple `print` statement. What if we misspelt the `print` as `Print`. Note the capitalization. In this case, Python *raises* a syntax error.

```
$ python
>>> Print 'hello world'
      File "<stdin>", line 1
        Print 'hello world'
                ^
SyntaxError: invalid syntax
```

Observe that a `SyntaxError` is raised and the location in the program where the error was detected is also printed. This is what the default *error handler* in Python does.

Let us take the case of taking an input from the user. Instead of entering some text, press Ctrl-D and see what happens.

```
$ python
>>> raw_input('Enter something : ')
Enter something : Traceback (most recent call last):
  File "<stdin>", line 1, in ?
EOFError
>>>
```

Python raises an error called `EOFError` which means that it found an *end-of-file* symbol (represented by Ctrl-D) when it was expecting some text.

Handling Exceptions

We can handle exceptions using the `try...except` statement. We put our usual block of statements within the `try`-block and put our error handlers in the `except`-block.

```
#!/usr/bin/env python
# File name: try_except.py

import sys

try:
    s = raw_input('Enter something : ')
except EOFError:
    print '\nWhy did you do an EOF on me?'
    sys.exit()
except:
    print '\nInvalid input.'
    # Here, we are not exiting the program
```

```
print 'Done'
```

Output

```
$ python try_except.py
Enter something : ^D
Why did you do an EOF on me?
```

```
$ python try_except.py
Enter something : abc
Done
```

How It Works

We put our block of statements into the `try`-block and then handle all the errors and exceptions in the `except`-block. The `except` clause can handle a specific error or exception, or a tuple of errors and exceptions. If no errors or exceptions are mentioned, it will handle *all* errors and exceptions. There has to be at least one `except` clause associated with every `try` clause.

If any error or exception is not handled, then Python will look for any exception handlers in the outer blocks, and if no exception handlers, it will call the default error handler which stops the execution of the program, prints the error message and then quits.

You can have an `else`-clause associated with a `try...except` clause, which will be executed if no exception occurs.

We can also receive the exception object in the `except` clause so that we can retrieve additional information about the exception which has occurred. This is demonstrated in the next example.

Raising Exceptions

You can *raise* exceptions using the `raise` statement. You also have to specify the name of the error/exception and the exception object that is to be *raised* along with the exception. The error or exception that you can raise should be a class which is directly or indirectly derived from the `Error` or `Exception` classes respectively.

```
#!/usr/bin/env python
# File name: raising.py

class ShortInputException(Exception):
    '''A user-defined exception class.'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast

try:
    s = raw_input('Enter something : ')
    if len(s) < 3:
        raise ShortInputException(len(s), 3)
    # Continue other work as usual here

except EOFError:
    print '\nWhy did you do an EOF on me?'
except ShortInputException, x:
    print 'Input was of length %d. I was expecting at least length of %d' %
(x.length, x.atleast)
else:
    print 'No exception was raised.'
```

Output

```
$ python raising.py
Enter something : abcd
No exception was raised.
```

```
$ python raising.py
Enter something : ab
Input was of length 2. I was expecting at least length of 3
```

How It Works

We are creating our own exception type, but we can use any built-in exception/error for *raising* errors such as `ValueError`, etc. Run `import exceptions; help(exceptions)` for details.

The new exception class we are creating is the `ShortInputException` class. It has two fields - `length` - which is the length of the given input and `atleast` - which is the minimum length that the program was expecting.

In the `except` clause, we mention the class of error as well as the variable to hold the corresponding error/exception object. This is analogous to parameters and arguments in a function call. In this particular `except` clause, we use the `length` and `atleast` fields of the exception object to print an appropriate message to the user.

Try...Finally

Suppose you had opened a file and were reading the file contents, and you want to close the file after processing, whether or not an exception was raised. This can be achieved using the `try...finally` statement. Note that you *cannot* use an `except` clause and a `finally` clause with the same `try` statement, you will have to embed one within another using two `try`-blocks if you want to have both exception handlers and *finalizers*.

Update: Python 2.5, released on September 19, 2006, allows a single `try` block to have both `except` and `finally` clauses.

```
#!/usr/bin/env python
# File name: finally.py

import time

f = file('poem.txt')

try:
    while True:
        line = f.readline()
        if len(line) == 0:
            break
        time.sleep(2)
        print line,
finally:
    f.close()
    print 'Closed the file.'
```

Output

```
$ python finally.py
Programming is fun
^C
```

```
Closed the file.  
Traceback (most recent call last):  
  File "programs/finally.py", line 13, in ?  
    time.sleep(2)  
KeyboardInterrupt
```

How It Works

We follow the usual file-reading idiom in this program, but we have introduced a delay of 2 seconds in-between each line reading using the `time.sleep` method. The reason to introduce this delay is so that the program runs slow enough for us to cancel the program while it is running.

When we process Ctrl-C when the program is running, a `KeyboardInterrupt` exception is raised and the `finally`-block is executed and then Python quits.

Summary

We have discussed how to use `try...except` and `try...finally` statements. We have seen how to create our own exception types and how to raise exceptions.

Next, we will explore the Standard Library.

Standard Library

Introduction

The Python Standard Library is available with every Python installation. It contains a huge number of useful modules. It is important that you become familiar with the Python Standard Library since most of our programs can be solved more easily if we can take advantage of these modules.

We will explore some of the commonly used modules in this library. You can find complete details for all of the modules in the [Python Standard Library documentation](#).

The os and sys modules

The `os` module contains operating-system specific functionality.

The `sys` module contains functionality related to the Python system and its environment. We have already used `sys.argv` which contains the command-line arguments passed to any Python program.

```
#!/usr/bin/env python
# File name: cat.py

import os, sys

def readfile(filename):
    '''Print a file to the standard output.'''
    if not os.path.exists(filename):
        print 'The file', filename, 'does not exist.'
        sys.exit(1)

    for line in file(filename):
        print line,

if len(sys.argv) < 2:
    print 'No action specified.'
    sys.exit()

if sys.argv[1].startswith('--'):
    option = sys.argv[1][2:]
    if option == 'version':
        print 'Version 1.5'
    elif option == 'help':
        print '''\
This program prints files to the standard output.
Any number of files can be specified.
Options include:
--version : Prints the version number
--help    : Display this help.
'''
    else:
        print 'Unknown option.'
        sys.exit()
else:
    for filename in sys.argv[1:]:
        readfile(filename)
```

Output

```
$ python cat.py
No action specified.

$ python cat.py --version
Version 1.5

$ python cat.py --help
This program prints files to the standard output.
Any number of files can be specified.
Options include:
  --version : Prints the version number
  --help    : Display this help.

$ python cat.py --nonsense
Unknown option.

$ python cat.py poem.txt
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

How It Works

This program tries to mimic the `cat` command familiar to Linux/BSD/Mac OS X command line users. You can specify the names of text files and the `cat` command will print all of them to the screen (standard output).

When a Python program is run (i.e. not in interactive mode), there is always at least one item in the `sys.argv` list which is the name of the current program being run and is available as `sys.argv[0]` since Python starts counting from 0. Other command line arguments follow this item.

To make the program user-friendly, we have supplied certain options that the user can specify to learn more about the program. We use the first argument to check if any options have been specified. If the `--version` option is used, the version number of the program is printed. Similarly, when the `--help` option is used, we give a bit of explanation about the program. We make use of the `sys.exit` function to quit the program. See `help(sys.exit)` for details.

When no options are specified and filenames are passed to the program, it simply reads out each file, one by one in the order specified on the command line.

The name `cat` is short for *concatenate* which is basically what the program does - it can print out files in order, in effect it can attach/concatenate files together in the output.

There are many other fields and methods that you can make use of in the `sys` module, such as `sys.version` and `sys.version_info` which gives you information about the version of Python running.

For experienced programmers, other items of interest in the `sys` module include `sys.stdin`, `sys.stdout` and `sys.stderr` which correspond to the standard input, standard output and standard error streams of your program respectively.

The `os` module can help your program be platform-independent i.e. written such that it can run on any system such as Linux or BSD or Mac OS X or Windows, etc. without requiring any changes. An example of how it can help is the `os.path.join` function that joins together directory filenames to give a path suitable to the operating system being used. The system-specific path operator can also be accessed as `os.sep`.

Some useful parts of the `os` module are listed below. Many of them are self-explanatory.

- The `os.name` string tells you which platform you are using, such as `nt` for Windows and `posix` for Linux/BSD/Mac OS X users.
- The `os.getcwd()` function gets the current working directory of the program.
- The `os.getenv()` function and the `os.putenv()` functions are used to get and set environment variables respectively. Alternatively, the `os.environ` dictionary can be used as well.
- The `os.listdir()` function returns the name of all the files and directories in the specified directory.
- The `os.remove()` function is used to remove a file.
- The `os.system()` function is used to run a shell command.
- The `os.path.split()` function returns the directory name and the file name of the given path.
- The `os.path.isfile()` and the `os.path.isdir()` functions allow us to check if the given path is a real file or directory respectively. Similarly, the `os.path.exists()` function can be used to check if the given path exists.

Summary

We have seen some of the functionality exposed by the `os` and `sys` modules of the standard library. There are a lot more modules available in the standard library that have a wide range of functionality.

Next, we will cover various aspects of Python that will make our tour of Python more interesting.

More Python

Introduction

We have covered many aspects of Python till now. In this chapter, we will cover some more aspects that will make our knowledge of Python more *complete* for daily practical purposes.

Special Methods

There are certain special methods which have special significance in classes such as `__init__`, `__del__` and `__str__` methods whose significance we have already seen.

Generally, special methods are used to mimic certain behavior. For example, if you want to allow the `x[key]` indexing operation for instances of your class, just like they are used for lists and tuples, then all you have to do is just implement the `__getitem__` method. When Python encounters `x[key]` for an instance of your class, it simply calls the `__getitem__` method of your class, and if you think about it, this is what Python does for the `list` class as well!

Some useful special methods are listed in the following table. If you want to know all the special methods, please see the [Special Method Names](#) section of the Python reference manual.

Name	Explanation
<code>__init__(self, ...)</code>	This method is used to initialize objects, and is called just before the newly created object is returned for usage
<code>__del__(self)</code>	Called just before the object is destroyed.
<code>__str__(self)</code>	Called when we need to <i>stringify</i> the object
<code>__lt__(self, other)</code>	Called when the <i>_less than_</i> operator (<) is used to compare this object with another object. Similarly, there are special methods for the relational operators (+, >, etc.)
<code>__getitem__(self, key)</code>	Called when <code>x[key]</code> indexing operation is used.
<code>__len__(self)</code>	Called when the built-in <code>len()</code> function is used for the sequence object.

Single statement blocks

By now, you should have firmly understood that each block of statements is set apart from the rest by its own indentation level. Well, this is true for the most part but there is one minor exception to the rule. If your block of statements contain only one single statement, then you can specify it on the same line of a conditional statement (or looping statement, etc.) The following example should make it clear.

```
$ python
>>> flag = True
>>> if flag == True: print 'Yes'
...
Yes
>>>
```

As we can see, the single statement is used in-place and is not written in a separate block. Although you can use this for making your program smaller, I *strongly* recommend that you do not use this short-cut method. This topic was included here for the sake of completeness.

Also notice that when the Python interpreter is used in interactive mode, it helps you enter the statements by changing the prompts appropriately. In the above case, after you have entered the `if` statement, it changes the prompt to `...` to indicate that it expects the statement to be continued. When we press enter/return key, Python then parses, compiles and runs the statement immediately. Then, it returns to the old prompt and waits for the next line of code.

List Comprehension

List comprehensions are used to derive a new list from an existing sequence. For example, you have a list of numbers and you want to get a corresponding list of numbers multiplied by 2 but only when the number is greater than 2. You could create an empty list, and then calculate these numbers and append to the list one-by-one, or you could take the simpler route and use list comprehensions.

```
$ python
>>> list_one = [2, 3, 4]
>>> list_two = [2*i for i in list_one if i > 2]
>>> list_two
[6, 8]
>>>
```

How It Works

We derive a new list by specifying the manipulation to be done ($2*i$) for each item of the original sequence (`for i in list_one`) when some condition is satisfied (`if i > 2`). The condition is optional.

Note that the original list is unmodified.

Receiving tuples and lists in functions

There is a special way of receiving parameters to a function as a tuple or a dictionary using the `*` or `**` prefix respectively. This is useful when we want to take variable number of arguments for the function.

```
$ python
>>> def sum(power, *args):
...     total = 0
...     for i in args:
```

```
...         total += i
...     return total
...
>>> sum(2, 3, 4)
7
>>> sum(2, 10)
10
>>>
```

Since we have the `*` prefix for the `args` variable, all extra arguments passed to the function are stored in the `args` variable as a tuple. If the `**` prefix was used instead, the extra parameters would be considered to be key-value pairs of a dictionary.

Lambda forms

A lambda statement is used to create new function objects and then use them at runtime. A lambda statement basically creates a nameless function.

```
$ python
>>> def make_repeater(n):
...     return lambda s: s * n
...
>>> twice = make_repeater(2)
>>> print twice('word')
wordword
>>> print twice(5)
10
>>>
```

How It Works

We use a function `make_repeater` to create new function objects and then return it. A lambda statement is used to create the function object.

Essentially, the `lambda` takes a tuple of parameters followed by a colon followed by a *single expression* which becomes the body of the function. This single expression is equivalent to a return statement with the same expression in a named function. Note that only expressions are allowed inside the 'body' of a lambda definition, not even the 'print' statement can be used.

The exec and eval statements

The `exec` statement is used to execute Python code stored in a string. For example, we can generate Python code at runtime and then execute these statements using the `exec` statement.

```
$ python
>>> exec 'print "Hello World"'
Hello World
>>>
```

The `eval` statement is used to evaluate Python expressions which are stored in a string.

```
$ python
>>> eval('2*3')
6
```

The assert statement

The `assert` statement is used to `_assert_` that something is true. For example, if you are sure that you will have a string with valid text in it, but want to check it *just in case*, and raise an `Error` if it is not true, then `assert` statement is ideal in this situation. When the `assert` statement sees that the condition is false, it raises an `AssertionError`.

```
$ python
>>> s = 'hello world'
>>> assert s is not None
>>> assert len(s) > 0
>>> assert len(s) == 0                # this should fail
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
>>>
```

The repr function

The `repr` function is used to obtain a string-ized representation of the object. Backticks (also called conversion or reverse quotes) can be used for the same things. Note that you will have `eval(repr(object)) == object` valid (at least most of the time).

```
$ python
>>> i = []
>>> i.append('item')
>>> `i`
"['item']"
>>> repr(i)
"['item']"
>>> eval(repr(i)) == i
True
>>>
```

You can control what the `repr` function gets by defining the `__repr__` method of your class.

Summary

We have covered some more features of Python in this chapter. We have still not covered all the features of Python. However, at this stage, we have covered most of what you will use on a daily basis. This is sufficient for you to get started with whatever problems you wanted to solve by learning programming. You can learn more details of Python as and when needed.