A Byte of Python

A Byte of Python

Swaroop C H
www.python.g2swaroop.net

Version 1.15

Revision History		
Revision 1.15	28/03/2004	g2
Minor revisions		
Revision 1.12	16/03/2004	g2
Additions and corrections		
Revision 1.10	09/03/2004	g2
More typo corrections, thanks to many enthusiastic and helpful readers.		
Revision 1.00	08/03/2004	g2
After tremendous feedback and suggestions from readers, I have made significant revisions to the content along with typo corrections.		
Revision 0.99	22/02/2004	g2
Added a new chapter on modules and about variable number of arguments in functions.		
Revision 0.98	16/02/2004	g2
Wrote a Python script and CSS stylesheet to improve XHTML output, including a crude lexical analyzer for syntax highlighting of the programs.		
Revision 0.97	13/02/2004	g2
Another completely rewritten draft, in DocBook XML (again). Book has improved a lot - It is more coherent and readable.		
Revision 0.93	25/01/2004	g2
Added IDLE talk and more Windows-specific stuff		
Revision 0.92	05/01/2004	g2
Changes to some examples, other improvements		
Revision 0.91	30/12/2003	g2
Corrected some typos, improvised certain topics		
Revision 0.90	18/12/2003	g2
Added 2 more chapters, OpenOffice format with revisions		
Revision 0.60	21/11/2003	g2

Fully Rewritten and Expanded		
Revision 0.20	20/11/2003	g2
Corrected some typos and errors		
Revision 0.15	20/11/2003	g2
Converted to DocBook XML		
Revision 0.10	14/11/2003	g2
Initial draft using KWord		

Abstract
The Perfect Anti-Venom for your programming problems!
This book will help you learn to use the Python language, whether you are new to computers or are an experienced programmer.

Dedication

This book is dedicated to each and every person in the free and open source software community. They work hard to create such wonderful software and technologies and share it with others. They selflessly share their knowledge and expertise with us. The efforts of all the programmers, designers, artists, documentation writers, bug-fixers and of course, the users, in creating such software and technologies is truly inspiring and amazing.

Table of Contents

List of Tables

List of Examples

Preface

Table of Contents

Python is probably one of the first programming languages out there which is both simple and powerful, which is good for both beginners as well as experts, and importantly, fun to program with. This book will help you learn this wonderful language and show how to get things done quickly and painlessly - in effect "The Perfect Anti-Venom for your programming problems!"

Who This Book Is For

This book serves as a guide or tutorial to the Python programming language. It is intended to help both newbies as well as experienced programmers to learn and get started with Python. If all you know about computers is how to save text files, then you can learn Python from this book. If you have previous programming experience, then you can learn Python from this book also.

If you do have previous programming experience, you will be interested in the differences between Python and your favorite programming language and I have highlighted many such differences for you. A little warning though, Python is soon going to become your favorite programming language!

History Lesson

I first started with Python when I needed to write an installer for my software Diamond so that I could make the installation easy. I had to choose between Python or Perl bindings for the Qt library. I did some research on the web and I came across an article where Eric S Raymond (affectionately called ESR) talked about how Python became his favorite programming language. I also found out that the PyQt bindings were very good. So I decided that Python was the language for me.

I then started searching for a good book on Python. I couldn't find any! I kept wondering why, though I found some O'Reilly books - they were either too expensive or were more like a reference manual than a guide. I settled for the documentation that came with Python, but it was too brief. It did give a good idea about Python but an

incomplete one and although I managed with it, I felt it was completely unsuitable for newbies.

About six months after my first brush with Python, I installed the (then) latest Red Hat 9.0 Linux and I was playing around with KWord when I suddenly got the idea of writing some stuff on Python. It quickly became 30 pages long and I became serious about converting it into a complete book. After many improvements and rewrites, it has reached this stage where it has become a useful and complete guide to learning the Python language. This book is my contribution and tribute to the open source community.

In the true spirit of open source, I have received lots of constructive suggestions and criticisms which have helped me improve this book a lot.

Status of this Book

This book is feature-complete as of today. However, your feedback is essential to improve this book.

More chapters are planned for the future, including using PyGTK/wxPython, the Python Imaging Library, and maybe even Boa Constructor.

Official Website

The official website of this book is www.python.g2swaroop.net . You can read the book online at this website. You can also download the latest versions of the book and send me feedback through this website.

License Terms

I'd Like To Hear From You

I have put in a lot of effort to make this book as interesting and as accurate as possible. However, if you find some things are not consistent or are incorrect, then

please do tell me so that I can correct them. Any constructive suggestions, appreciation and criticisms can be sent to me at python@g2swaroop.net .

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

Chapter 1. Introduction

Table of Contents

Introduction

Python is one of those rare languages which can claim to be both simple and powerful. You will be pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than on the syntax (i.e. the structure of the program that you are writing) of the language.

The official Python introduction 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.

I will discuss these features in more detail in the next section.

By the way, 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 (but very strict English!). This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the syntax i.e. the language itself.

Easy to Learn

As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax as already mentioned.

Free and Open Source

Python is an example of a FLOSS (Free/Libre and Open Source Software). In simple terms, you can freely distribute copies of this software, read the software's source code, make changes to it, use pieces of it in new free programs, and that you know you can do these things. FLOSS 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 improved by a community who just want to see a better Python.

High-level Language

When you write programs in Python, you never need to bother about low-level details such as managing the memory used by your program.

Portable

Due to its open-source nature, Python has been ported (i.e. changed to make it work on) to many many platforms. All your Python programs will work on any of these platforms without requiring any changes at all. However, you must be careful enough to avoid any system-dependent features.

You can use Python on Linux, Windows, Macintosh, 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 and PocketPC !

Interpreted

This requires a little explanation.

A program written in a compiled language like C or C++ is translated from the source language i.e. C/C++ into a language 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 just stores the binary code in the computer's memory and starts executing from the first instruction in the program.

When you use an interpreted language like Python, there is no separate compilation and execution steps. You just run the program from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then runs it using a virtual machine. All this makes using Python so much easier. You just run your programs - you never have to worry about linking and loading with libraries, etc. They are also more portable this way because you can just copy your Python program into another system of any kind 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 programs. In object-oriented languages, the program is built around objects

which combine data and functionality. Python has a very powerful but simple way of doing object-oriented programming, especially, when compared to languages like C++ or Java.

Extensible

If you need a critical piece of code to run very fast, you can achieve this by writing that piece of code in C, and then combine that with your Python program.

Embeddable

You can embed Python within your C/C++ program 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 also other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the "batteries included" philosophy of Python.

Besides the standard library, there are various other high-quality libraries such as the Python Imaging Library which is an amazingly simple image manipulation library.

Summary

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

Why not Perl?

If you didn't know, Perl is another very popular, free and open source interpreted language.

If you have ever tried writing a large program in Perl, you would have answered this question yourself! Perl programs are easy when they are small and simple, but they quickly become unwieldy once you start writing bigger programs. When compared with Perl, programs written in Python are definitely simpler, clearer, easier to write and hence more understandable and maintainable. I do admire Perl and I still use it (especially for my website), but whenever I write a program, I always think in terms of Python. Too much hacking and changes have made Perl very complicated and confusing. However, the upcoming Perl6 might do something about this but that is still a long way off.

The only significant advantage that Perl has is its library called CPAN - the Comprehensive Perl Archive Network. As the name suggests, this is a humongous collection of Perl modules and 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 a better library than Python is that it has been around for a much longer time than Python.

Also, the new Parrot virtual machine is designed to run both the completely redesigned Perl6 as well as Python and other interpreted languages like Ruby, PHP and Tcl. What this means to you is that maybe you will be able to use all Perl modules from Python in the future, so that you will have the best of both worlds - the powerful CPAN library combined with the powerful Python language. However, we will just have to wait and see what happens.

What Programmers Say

You may find it interesting to read what great programmers like ESR have to say about Python.

- Eric S Raymond (of "The Cathedral and the Bazaar" fame and also the person who coined the term "open source") says that Python has become his favorite language. Read more at Linux Journal . This article was the real inspiration for my first brush with Python.

- Bruce Eckel (of "Thinking in C++" and "Thinking in Java" fame) says that no language has made him more productive than Python. He says that Python is the only language that focuses on making things easier for the programmer. Read the complete interview at Artima.com . This interview/article is Part 1 of a four-part series of interviews called "Python and the Programmer - A Conversation with Bruce Eckel").

- Peter Norvig, a well-known Lisp author (thanks to Guido van Rossum for pointing that out) and Director of Search Quality at Google says that Python has always been an integral part of Google. You can actually verify this statement by looking at the Google Jobs website which lists Python as a requirement for software engineers.

- Bruce Perens (co-founder of OpenSource.org along with ESR) has started an initiative called "UserLinux" which aims to create a standardized Linux distribution supported by multiple vendors. Python has beaten contenders like

Perl and Ruby, to become the main programming language that will be supported by UserLinux.

Chapter 2. Installing Python

Table of Contents

For Linux/BSD Users

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

To test if you have Python already installed in your Linux box, open a shell terminal (gnome-terminal or konsole or xterm) and enter the command python -V as shown below.

```
$ python -V
Python 2.3.3
```

> **Note**
> $ is the prompt of the shell. It will be different for different users and hence I will indicate the prompt by just $.

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 one:

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

then you don't have Python installed. This is highly unlikely but possible. You have two ways of installing Python on your system.

- Install Python from your Linux distribution CDs [1] such as your Fedora/Red Hat/Mandrake/Debian/{your choice here}.

- Visit Python.org/download/ , get the latest version and follow the instructions given at the website on how to install Python.

For Windows Users

Visit Python.org/download/ and download the latest version from this website (which was Python-2.3.3.exe as of this writing). It is about 9 MB only. It is very compact compared to other languages. Installation is just like any other Windows-based software.

> **Caution**
> When you are given the option of unchecking any optional components, please don't! Some of these components will be useful for you, especially IDLE (short for Integrated DeveLopment Environment).

An interesting fact is that about 70% of Python downloads are by Windows users. Of course, this doesn't give the complete picture since almost all Linux users will have Python installed already on their systems by default.

> **Note**
> You can always use IDLE to run your Python programs, but if you want to run your Python programs from the DOS prompt, then add the following line to C:\AUTOEXEC.BAT :

```
PATH=%PATH%;C:\Python23
```

> Then restart the system. However, be sure to give the correct folder name.
> A reader has also pointed out to me that Windows XP requires using the menu Control Panel -> System to set environment variables like we did above.

Summary

It is easy to install Python on Windows and will probably be already installed on your Linux/BSD system or you may have had to install Python by yourself - whichever the case, I will assume that you have Python installed on your system now.
Next, we will write our first Python program.

[1] See LinCDs.com for more details.

Chapter 3. First Steps

Table of Contents

Introduction

We will now see how to run a traditional "Hello World" program in Python. This will also 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

Run the interpreter interactively by entering python in the shell. Now enter print 'Hello World' followed by the Enter key. You should see the words Hello World as output.

> **Caution**
> When I mean run the interpreter, Linux users can use the shell as shown in the examples here or use IDLE. Windows users can click on Start -> Programs -> Python 2.3 -> IDLE (Python GUI). The Programs menu folder name might be different depending on the version of Python that you have installed.

Example 3.1. Using The Python Interpreter prompt

```
$ python
Python 2.2.3 (#1, Oct 15 2003, 23:33:35)
[GCC 3.3.1 20030930 (Red Hat Linux 3.3.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>>
```

What you just entered is a single Python statement i.e. you stated to Python that you want to get something done. We use the print operator to (unsurprisingly) print any

value that you give it. Here, we are supplying the string Hello World to the print statement which promptly prints it to the screen.

> **Important**
> To exit the Python interpreter prompt, press
>
> Ctrl-d if you are using Linux or using IDLE (both Linux and Windows).
>
> Ctrl-z followed by Enter if you are using Windows, especially, the DOS prompt.

Choosing an Editor

Before we get started on writing Python programs in a source file, we need an editor to create the source files. The choice of an editor is very crucial indeed. 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 helps you reach your destination (achieve your goal) in a much faster and safer way.

One of the basic requirements is syntax highlighting where the Python keywords, operators, strings, etc. i.e. the things that are special to Python are colorized so that you can see your program and visualize its running.

If you are using Windows, then I suggest that you stick with IDLE. IDLE does syntax highlighting and a lot more such as allowing you to run your programs from within IDLE and so on. Whatever you do, don't use Notepad - because, among other things, we will need indentation and it would be tedious to type it in Notepad compared to other good editors such as IDLE which will automatically help you with this.

If you are using Linux/FreeBSD, then I suggest that you use KWrite or you can use IDLE as well. If you are an experienced programmer, then you must be already using VIM or Emacs or XEmacs. Even Microsoft people use VIM and Emacs while demoing their .NET programs at their famous Professional Developers' Conference (PDC). VIM and XEmacs are available for Windows and are always part of the standard Linux and BSD systems.

I personally use VIM for most of my programs but occasionally I do use XEmacs. These two programs are the most powerful editors you can ever find. If you intend to do a lot of programming or even editing, then I highly recommend that you learn one of these editors. There are tutorials that come with these editors to help you get started.

If you still want to explore other choices, please see the comprehensive list of Python Editors at Python.org - take a look and make your choice. I repeat, please choose a proper editor - it will help you a lot in the long run.

Using A Source File

Using a Source File

Okay, now let's get back to some programming. We will write the traditional "Hello World" program - whenever you learn a new programming language, the first program that you write and run is usually a Hello World program. As Simon Cozens (one of the leading Perl6/Parrot hackers) puts it, it is the traditional incantation to the programming gods to help you learn the language better.

Open your editor of choice, enter the following program and save it as hello_world.py. All Python programs usually have the file extension .py .

Example 3.2. Using a Source File

```
#!/usr/bin/python
# Filename : hello_world.py
print 'Hello World!'
```

Run this program by opening a shell (Linux terminal or DOS prompt) and entering the command python hello_world.py. If you are using IDLE, use the menu Edit -> Run Script or the keyboard shortcut Ctrl-F5. The output is as shown below.

Output

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

If you got the above output, congratulations! You have successfully run your first Python program. If you got an error, please type the program exactly as above and run the program again. Note that Python is case-sensitive i.e. print is not the same as Print - note the lowercase p in the former and the uppercase P in the latter. Also, ensure there are no spaces or tabs before the first character in each line - we will see later why this is important.

How It Works

Consider the first two lines:

```
#!/usr/bin/python
# Filename : hello_world.py
```

These are called comments. Anything to the right of the # character (which is not inside a string) is a comment and is mainly useful as notes for the reader of the program. Python does not use comments in any way.

However, the first line in this case is special. It is called the shebang line. Whenever the first two characters of the source file are #! - followed by the location of the interpreter it tells your Linux/Unix system that this program should be run with this interpreter, when you execute the program. This is explained in detail in the next section. Note that you can always run the program on any platform using the interpreter directly by running the command python program.py .

Important
Please use comments sensibly in your program so that readers of the program can easily understand the program when they read it. Remember, that person could be yourself after six months!

Executable Python programs

This applies only to Linux/Unix systems but Windows users might be curious as well about the first line of the program. First, give the program executable permission using the chmod command and then run the source program.

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

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

To make things more fun, you can rename the file to just hello and run it as ./hello and it will still work since the system knows that it has to run the program using the interpreter located at the filename specified in the first line.

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

Whenever you run a program, the system looks for that command in each of the directories listed in the PATH environment variable.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin
$ cp hello /home/swaroop/bin
$ hello
Hello World!
```

What we are doing here, is that we first see what the PATH environment variable contains using the echo command as shown above. Note that we retrieve the value of variables in the shell by prefixing $ to the name of the variable. We see that /home/swaroop/bin is one of the directories in the PATH variable (where swaroop is the username I am using in my system). There might be a similar directory for your username in your system. Next, we copy the file to this directory. Now, we simply run hello and we get those famous words. Note that now you can run your program from anywhere i.e. irrespective of your current directory.

This method will be very useful when you want to write certain scripts and you want to be able to run those programs anytime anywhere. It is like creating your own commands just like cd or any other commands you use in the Linux terminal or DOS prompt.

Caution
With respect to Python, a program or script or software all mean the same thing.

Getting Help

If you need quick information about any function or statement in Python, then use the help functionality. This is especially useful when using the interpreter prompt. For example,

```
>>> help(str)
```

This displays the help for the str class which represents all strings that you use in your program. Try this now. Classes will be explained in detail in the chapter on object-oriented programming.

> **Important**
> Press q to exit the help.

Similarly, you can obtain information about almost anything in Python. Use help() to learn more about using help itself! However, to get help for operators like print, you need the PYTHONDOCS environment variable set. This can be done easily on Linux/Unix using the env command.

```
$ env \
> PYTHONDOCS=/usr/share/doc/python-docs-2.2.3/html/ \
> python
Python 2.2.3 (#1, Oct 15 2003, 23:33:35)
[GCC 3.3.1 20030930 (Red Hat Linux 3.3.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> help('print')
```

You will notice that I have used quotes to specify print as a string - this is so that Python can understand that it does not have to print something, but it should fetch help about the print statement. The location that I have used above is the one for the Python standard installation which comes with Fedora Core 1 Linux, it might be different for different installations of Python.

Summary

You should now be able to write, save and run Python programs at ease. Now that you are a Python user, let's learn some more Python concepts - the ones that get the job done.

Chapter 4. The Basics

Table of Contents

Just printing "Hello World" is not exciting, is it? You want to do more than that - you want to be able to do some manipulation of data and get some output just like you would with other commands on your system. We can achieve this using constants and variables.

Literal Constants

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 - you use its value literally. The number 2 always represents itself and nothing else. It is a constant because it's value cannot be changed.

Numbers

Numbers in Python are of four types - integers, long integers, floating point and complex numbers. Examples of integers are 2 and -3 which are just whole numbers. Long integers are just bigger numbers. Examples of floating point numbers (or floats for short) 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

A string is a sequence of characters . Strings are basically just words.
I can almost guarantee that you will be using strings in every Python program that you write, so pay careful attention to the following part. Here's how you can use strings in Python:

- Using 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.

- Using Double Quotes("). Strings in double quotes work exactly the same way as strings in single quotes. An example is "What's your name?" .

- Using Triple Quotes (''' or """). You can specify multi-line strings using triple quotes. Also, you can use single quotes and double quotes freely within triple quotes. Examples are

```
'''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 string is What's your name?. You cannot specify 'What's your name?' because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string. This is done with the help of what is called an escape sequence. You specify 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 to use an escape sequence for using a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using an 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 an escape sequence for the newline character \n which indicates that a new line is about to start. An example is 'This is the first line.\nThis is the second line.'

  Other escape sequences include \r (carriage return), \a (bell), etc. Also, remember that a single backslash at the end of the line indicates that the string is continued in the next line but it does not mean that there is a newline. For example,

```
"This is the first sentence.\
This 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 you don't want any special processing such as escape sequences, then you can specify the string as a raw string by prefixing `r` or `R` to the string. An example is `r"Newlines are indicated by \n."`.

- Unicode Strings. Unicode is a standard used for internationalization. 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. If you want to use Unicode strings in Python, you can prefix the string with `u` or `U` such as `u"This is a Unicode string."`. Remember to use Unicode strings when you deal with text files, especially, if it involves 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 isn't. We will see why this is not a limitation in the various programs that we 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?"` .

Note for C/C++ Programmers

There is no separate `char` data type in Python. There is no real need for it and I am sure you won't 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.

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'`.

Variables

Using just literal constants can soon become boring - we need some way of storing some information and manipulate that information. This is where variables come into the picture. Variables are exactly what they mean - their value can vary i.e. you can store anything in a variable. Variables are just parts of your computer's memory where

you store some information. Unlike literal constants, you need some method of accessing these variables i.e. you give them names.

Identifier Naming

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

- The first character of the identifier must start with a letter of the alphabet (upper or lowercase) or an underscore ('_').
- The rest of the identifier name can consist of letters, underscores or digits.
- Identifier names are case-sensitive. For example, myname and myName are not the same. Note the lowercase n in the former and the uppercase N in the latter.
- Examples of valid identifier names are i, __my_name, name_23, and a1b2_c3.
- Examples of invalid identifier names are 2things, this is spaced out and 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. We will see how to create your own types using classes in the chapter on object-oriented programming.

Objects

Remember, Python refers to anything used in a program as an object. This is meant in the generic sense. Instead of saying "the something that we have", we say "the object that we have" .

> **Note to Object Oriented Programming users**
> Python is strongly object-oriented in the sense that everything is an object including numbers, strings and even functions.

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

> **Note**

Henceforth, the standard procedure to save and run your Python program is as follows.

1. Open your favorite editor.
2. Enter the program code given in the example.
3. Save it as a file with the filename mentioned in the comment. All Python programs should have an extension of .py .
4. Run the interpreter with the command python program.py or use IDLE to run the programs. You can also use the executable method as explained earlier.

Example 4.1. Using Variables and Literal Constants

```
#!/usr/bin/python
# Filename : var.py

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

s = '''This is a multi-line string.
This is the second line.'''
print s
```

Output

```
$ python var.py
5
6
This is a multi-line string.
This is the second line.
```

How It Works

First, we assign the literal constant value 5 to the variable i using the assignment operator (=). This line is called a statement because it states that something should be done. Next, we print the value of i using the print statement.
Then, we add 1 to the value stored in i and store it back in i and then we print the value to confirm that it is indeed 6.

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

Note to C/C++ Programmers
Variables are used by just naming them and assigning a value. No declaration or data type definition is required.

Logical and Physical Lines

Python implicitly assumes that each physical line corresponds to a logical line. A physical line is what you see when you write the program. A logical line is what Python sees as a single statement. An example of a logical line is a statement like print 'Hello World!'. If this was on a line by itself, it also corresponds to a physical line.
If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using the semicolon (;) which indicates the end of a logical line/statement. For example,

```
i = 5
print i
```

is effectively equal to

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

The neat thing is that you don't need to put the semicolon if you write a single logical line in every physical line. Experienced programmers need to remember this in particular.

The above two Python statements can also 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 only. Use more than one physical line only if the logical line is really long. Avoid using the semicolon as far as possible. In fact, I have never used or even seen a semicolon in a Python program and this makes the programs simpler and more readable.

An example of writing a logical line spanning many physical lines follows. This is referred to as explicit line joining.

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

This gives the output

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

Similarly,

```
print \
i
```

is equivalent to

```
print i
```

Sometimes, there is an implicit assumption in certain logical statements spanning multiple physical lines where you don't need to use the backslashes. These are statements which involve parentheses, square brackets or curly braces. This is called implicit line joining. You can see this when we write programs using lists in later chapters.

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 indentation. Each such set of statements is called a block. We will see examples of how blocks are important in later chapters.

One example of how wrong indentation can give rise to errors is

```
i = 5
 print 'Value is', i              # Notice a single space at the start of the line.
# This is an error!
```

```
print 'I repeat, the value is', i
```

When you run this, you get the following error:

```
  File "<stdin>", line 2
    print i
    ^
SyntaxError: invalid syntax
```

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. This means that you cannot arbitrarily start new blocks of statements (except for the main block which you have been using all along, of course). Cases where you can use new blocks will be detailed in later chapters such as the control flow chapter.

Important
Do not use a mixture of tabs and spaces for the indentation as it is not cross-platform compatible. I strongly recommend that you use a single tab or two or four spaces for each indentation level. Choose one of these three indentation styles and use it consistently i.e. use that indentation type only.

Summary

Now that we have gone through the nitty-gritty details, we can move on to more interesting stuff such as control flow statements. Be sure to become comfortable with what you have read in this chapter.

Chapter 5. Operators and Expressions

Table of Contents

Introduction

Most statements (logical lines) that you write will contain expressions. A simple example of an expression is `2 + 3`. An expression can be broken down into operators and operands. 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. In this case, `2` and `3` are operands.

Operators

We will briefly take a look at the operators and their usage, depicted in the following table.

Tip

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

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Table 5.1. Operators and their usage

Operator	Name	Explanation	Examples
+	Plus	Adds the two objects	3 + 5 gives 8. 'a' + 'b' gives 'ab'.
-	Minus	Either gives a negative number or gives the subtraction of one number from the other.	-5.2 returns a negative number. 50 - 24 gives 26.
*	Multiply	Gives the multiplication of the two numbers or returns the string repeated by that many times.	2 * 3 gives 6. 'la' * 3 gives 'lalala'.
**	Power	Returns x to the power of y	3 ** 4 gives 81 (i.e. 3 * 3 * 3 * 3)
/	Divide	Divides x by y	4/3 gives 1 since 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 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 (binary digits i.e. 0 and 1)	2 << 2 gives 8. 2 is represented by 10 in bits. Left shifting by 2 bits gives 1000 which represents the decimal 8.
>>	Right Shift	Shifts the bits of the number to the right by the number of bits specified.	11 >> 1 gives 5. 11 is represented in bits by 1011 which when right shifted by 1 bit gives 101 which is nothing but decimal 5.
&	Bit-wise AND	Bitwise AND of the numbers	5 & 3 gives 1.
|	Bit-wise OR	Bitwise OR of the numbers	5 | 3 gives 7.
^	Bit-wise XOR	Bitwise XOR of the numbers	5 ^ 3 gives 6
~	Bit-wise invert	The bit-wise inversion of x is -(x+1)	~5 gives -6

Operator	Name	Explanation	Examples
<	Less Than	Returns whether x is less than y. All comparison operators return 1 for true and 0 for false. This is equivalent to the special variables True and False respectively. Note the capitalization of these variables' names.	5 < 3 gives 0 and 3 < 5 gives 1 i.e. 5 < 3 gives False and 3 < 5 gives True. Comparisons can be chained arbitrarily. 3 < 5 < 7 gives True.
>	Greater Than	Returns whether x is greater than y	5 > 3 returns True. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns False.
<=	Less Than or Equal To	Returns whether x is less than or equal to y	x = 3; y = 6; x <= y returns True.
>=	Greater Than or Equal To	Returns whether x is greater than or equal to y	x = 4; y = 3; x >= 3 returns True.
==	Equal To	Compares if the objects are equal	x = 2; y = 2; x == y returns True. x = 'str'; y = 'stR'; x == y returns False. x = 'str'; y = 'str'; x == y returns True.
!=	Not Equal To	Compares if the objects are not equal	x = 2; y = 3; x != y returns True
not	Boolean NOT	If x is True, it returns False. If x is False, it returns True	x = True; not y returns False.
and	Boolean AND	x and y returns False if x is False, else it returns evaluation of y	x = False; y = True; x and y returns False since x is False. In this case, Python will not evaluate y since it knows that the value of the expression has to be False. This is called short-circuit evaluation.
or	Boolean OR	If x is True, it returns x else it returns evaluation of y	x = True; y = False; x or y returns True. Short-circuit evaluation applies here as well.

Operator Precedence

If you had an expression such as 2 + 3 * 4, is the addition done first or the multiplication? Our high school maths tells us that the multiplication should be done

first. It means that the multiplication operator has higher precedence that 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 an expression, Python will first evaluate the operators lower in the following table before the operators listed higher in the table.

I have given the following list for the sake of completeness. I strongly advise you to use parentheses for grouping of operators and operands in order to explicitly specify the precedence and to make the program as readable as possible. For example, `2 + (3 * 4)` is definitely easier to understand than `2 + 3 * 4`. As with everything else, the parentheses should be used sensibly.

Table 5.2. Operator Precedence

Operator	Description
lambda	Lambda Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and Subtraction
*, /, %	Multiplication, Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation
x.attribute	Attribute reference
x[index]	Subscription
x[index:index]	Slicing
f(arguments, ...)	Function call
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key:datum, ...}	Dictionary display
`expressions, ...`	String conversion

The operators which are not already explained will be seen in later chapters.

Operators with the same precedence are listed in the same row in the above table. 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 they are evaluated, you can use parentheses. For example, if you want the addition to be evaluated before multiplication in an expression like `2 + 3 * 4` , then write the expression as `(2 + 3) * 4`.

Associativity

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

Expressions

Using Expressions

Example 5.1. Using Expressions

```
#!/usr/bin/python
# Filename : 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

Here, we are given the length and breadth of a rectangle. We then calculate the area and perimeter of the rectangle using 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 clean nice output and the program is much more readable this way. This is an example of how Python makes life easy for the programmer.

Summary

We have seen how to use operators, operands and expressions - these are the basic building blocks of any program. Next, we will see how we make use of these in our programs using other statements.

Chapter 6. Control Flow

Table of Contents

Introduction

In the programs we have seen till now, there has always been a series of statements and Python faithfully executes them in the same order. What if you wanted to change the flow of execution? For example, to take some decisions and do different things depending on different situations such as printing "Good Morning" or "Good Evening" depending on the time of the day?

As you might have guessed, this is achieved using control flow statements. There are three control flow statements in Python - if, for and while.

The if statement

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

Using the if statement

Example 6.1. Using the if statement

```
#!/usr/bin/python
# Filename : 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 don't win any prizes!)" # new block ends here
```

```
elif guess < number:
    print 'No, it is a little higher than that.' # another block
    # You can do whatever you want in a block ...
else:
    print 'No, it is a little lower than that.'
    # you must have guess > number to reach here
print 'Done'
# This last statement is always executed, after the if statement
# is executed.
```

Output

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that.
Done
$ python if.py
Enter an integer : 22
No, it is a little higher than that.
Done
$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you don't 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. Functions are just reusable pieces of programs.

We supply a string to the built-in raw_input function which then prints it to the screen and waits for input. Once we enter a number and press enter, then the function returns that input which, in the case of raw_string(), is always a string. We then convert this string to an integer using int and then store it in the variable guess. Actually, the int is a class but all you need to know right now is that you can use it to convert a string to an integer.

Then, we compare the guess of the user with the number we have. 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 'one tab per indentation level' rule.

Notice how the if statement contains a colon at the end - we are indicating to Python that a block of statements follows.

Then, we check if the guess is less than the number, and if so, we inform the user to guess a little higher than that. What we have used here is the elif clause which actually combines two related if else-if else statements into one combined if-elif-else statement. This makes the program more readable and clearer. It also 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 higher level of indentation, of course).

You can have another if statement inside the if-block of an if statement - this is called a nested if statement.

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 execution of your program starts and Python moves on to the print 'Done' statement, then sees the end of the program and quits.

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. All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds) and requires you to become aware of them initially but after that, you will become comfortable with it.

Equivalent C program

We will now see an equivalent program of the above in C to help newbies to understand how easy Python is compared to a language like C, and to help experienced programmers grasp the differences between the C/C++ family of languages and Python.

You can skip this section if you want to.

Note that indentation does not matter in C. One of the reasons why C programs are (usually) more difficult to understand is that it may not be written clearly. However, a good programmer always has a good and consistent indentation style. When it comes to Python, the programs always have to be written clearly!

Example 6.2. Equivalent C Program

```
#include <stdio.h>
/* Filename: if.c */

/* Execution of a C program always start from the main() function */
int main()
{
    /* declare variables before using them,
       we also have to specify the data type of the variables.
    */
    int number, guess;

    /* the number we have to guess */
    number = 23;

    /* input the guess of the user */
    printf("Enter an integer : ");
    scanf("%d", &guess);

    if (guess == number) /* expression within parentheses */
    {                    /* block enclosed within parentheses */
        printf("Congratulations, you guessed it.\n");
        printf("(but you don't win any prizes!)\n");
    }
    else
        if (guess < number)
        {
            printf("No, it is a little higher than that.\n");
        }
        else
        {
            /* you must have guess > number to reach here */
            printf("No, it is a little lower than that.\n");
        }

    printf("Done.\n");

    return 0; /* return an exit value to the shell */
}
```

Note to 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. A while statement can have an optional else clause.

Using the while statement

Example 6.3. Using the while statement

```
#!/usr/bin/python
# Filename : while.py

number = 23
stop = False

while not stop:
        guess = int(raw_input('Enter an integer : '))

        if guess == number:
                print 'Congratulations, you guessed it.'
                stop = True # This causes the while loop to stop
        elif guess < number:
                print 'No, it is a little higher than that.'
        else: # you must have guess > number to reach here
                print 'No, it is a little lower than that.'
else:
        print 'The while loop is over.'
        print 'I can do whatever I want here.'

print 'Done.'
```

Output

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
I can do whatever I want here.
Done.
```

How It Works

Here, 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 repeated guesses. This aptly demonstrates the use of the while statement.

We move the raw_input and if statements to inside the while loop and set the variable stop to True before the while loop. First, we check the variable stop and if it is True, we proceed to execute the corresponding while-block. After this block is executed, the condition is again checked which in this case is the stop variable. If it is true, we execute the while-block again, else we continue to execute the optional else-block if it exists, and then continue to 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 the condition is checked. If there is an else clause for a while loop, it is always executed unless you have a while loop which loops forever without ever breaking out!

The True and False are just special variables which are assigned the value 1 and 0 respectively. Please use True and False instead of 1 and 0 wherever it makes more sense, such as in the above example.

The else-block is actually redundant since you can put those statements in the same block (as the while statement) after the while statement. This has the same effect as an else-block.

Note to C/C++ Programmers
Remember that you can have an else clause for the while loop.

The for loop

The for..in statement is another looping statement where the number of times that the loop is executed is known (one way or another). The for statement is used to iterate over the elements of a sequence i.e. go through each item of a sequence. We will see more about sequences in detail in later chapters. What you need to know right now is that a sequence is just an ordered collection of items.

Using the for statement

Example 6.4. Using the for statement

```
#!/usr/bin/python
# Filename : for.py

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

```
else:
    print 'The for loop is over.'
```

Output

```
$ python for.py
1
2
3
4
The for loop is over.
```

How It Works

In this program, we are printing a sequence of numbers. We get this sequence of numbers using the built-in range function. What we do here is supply it two numbers and range returns a sequence of numbers starting from the first number and up to the second number. For example, range(1, 5) returns the sequence [1, 2, 3, 4]. By default, range takes a step count of 1. If we supply a third number to range, then that becomes the step count.

For example, range(1, 5, 2) returns [1, 3]. Remember, that the range extends up to the second number i.e. it does not include the second number.

The for loop then iterates over this range i.e. for i in range(1, 5) is equivalent to for i in [1, 2, 3, 4] which is like assigning each number in the sequence to i one at a time and executing the block of statements for each value of i. In this case, we print the value of i.

Remember that the else part is optional. When included, it is always executed once after the for loop is over.

Remember that the for..in loop works for any sequence. Here we have used a list generated by the built-in range function but in general, we can use any list, tuple or string. We will explore this in detail in later chapters.

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

The Python for loop is radically different from the C/C++ for loop. C# programmers will note that the for loop in Python is similar to the foreach loop in C#. Java programmers will note that the same is similar to for(int i : IntArray) in Java 1.5 .

In C/C++, if you write `for (int i = 0; i < 5; i++)`, then in Python, you write `for i in range(0, 5)`. As you can see, the `for` loop is simpler, more expressive and less error prone in Python.

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 been completely iterated over.

An important note is that if you break out of a for or while loop, any loop `else` block is not executed.

Using the break statement

Example 6.5. Using the `break` statement

```
#!/usr/bin/python
# Filename : break.py

while True:
        s = raw_input('Enter something : ')
        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 22
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 need to take care of the special condition where if the user enters the string quit, then we have to stop the loop. This is done by checking if the input is equal to the string quit and if so, then we use the break statement to quit the loop. Remember that the break statement can be used with the for loop as well.

G2's Poetic Python

The input I have used here is a mini poem I have written called G2's Poetic Python:

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

The continue statement

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

Using the continue statement

Example 6.6. Using the continue statement

```
#!/usr/bin/python
# Filename : continue.py

while True:
        s = raw_input('Enter something : ')
        if s == 'quit':
                break
        if len(s) < 3:
                continue
        print 'Sufficient length'
```

Output

```
$ python continue.py
Enter something : a
Enter something : 12
Enter something : abc
Sufficient length
Enter something : quit
```

How It Works

In this program, we accept input from the user, but we process them only if they are at least 3 characters long. So, we use the built-in `len` function which gives the length of the string. If the value returned by the `len` function is less than 3, then we skip the rest of the statements in the block using the `continue` statement, otherwise the rest of the statements in the loop are 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. These are the most often used parts of Python, so becoming comfortable with them is essential. Next, we will see how to create and use functions.

Chapter 7. Functions

Table of Contents

Introduction

Functions are reusable pieces of programs. They allow you to give a name to a block of statements and you can execute that block of statements by just using that name, anywhere in your program and any number of times. This is known as calling the function.

Functions are defined using the def keyword. This is followed by an identifier name for the function. This is followed by a pair of parentheses which may enclose some names of variables. The line ends with a colon and this is followed by a new block of statements which forms the body of the function. An example will make this easy to understand.

Defining a Function

Example 7.1. Defining a function

```
#!/usr/bin/python
# Filename : function1.py

def sayHello():
    print 'Hello World!' # A new block
# End of the function
sayHello() # call the function
```

Output

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

How It Works

We define a function called sayHello using the syntax explained above. This function takes no parameters - there are no variables declared in the parentheses. We call this function by specifying the name of the function followed by a pair of parentheses.

Function Parameters

A function can take parameters. Parameters are just values you supply to the function so that the function can do something by utilising those values. 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 function itself.

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

Using Function Parameters

Example 7.2. Using Function Parameters

```
#!/usr/bin/python
# Filename : func_param.py

def printMax(a, b):
    if a > b:
        print a, 'is maximum'
    else:
        print b, 'is maximum'

printMax(3, 4) # Directly give literal values

x = -5
y = -7

printMax(x, y) # Give variables as arguments
```

Output

```
$ python func_param.py
4 is maximum
-5 is maximum
```

How It Works

Here, we define a function called printMax where we take two parameters called a and b. We print the greater number using an if statement. In the first usage of printMax, we directly supply the numbers i.e. the arguments. In the second usage, we call the function using variable names. printMax(x, y) causes value of argument x to be assigned to parameter a and value of argument y to be assigned to parameter b. The printMax function works the same either way.

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function. That is, variable declarations 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.

Using Local Variables

Example 7.3. Using Local Variables

```
#!/usr/bin/python
# Filename : func_local.py

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

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

Output

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

How It Works

The variable x that we define is local to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected.

Using the global statement

If you want to assign to a variable defined outside the function, then you have to use the global statement. This is used to declare that the variable is global i.e. it is not local. It is impossible to assign to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (and there is no variable with the same name within the function). However, this is discouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the global statement makes it clear that the variable is defined in an outer block.

Example 7.4. Using the global statement

```
#!/usr/bin/python
# Filename : func_global.py

def func():
    global x

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

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

Output

```
$ python func_global.py
x is 50
Changed 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 to x inside the function, that change is reflected when we use the value of x in the outer block i.e. the main block in this case.

You can specify more than one global variable using the same global statement. For example, global x, y, z .

Default Argument Values

For some functions, you may want to make some 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 argument.

Note that the default argument value should be immutable. This may not make much sense now but you will understand it when you come to the later chapters. Just remember that you have to use only immutable values and you cannot use mutable objects such as lists for default argument values.

Using Default Argument Values

Example 7.5. Using Default Argument Values

```
#!/usr/bin/python
# Filename : func_default.py

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

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

Output

```
$ python func_default.py
Hello
WorldWorldWorldWorldWorld
```

How It Works

The function named say is used to print a string as number of times as we want. If we don't supply a value, then by default, the string is printed just once. This is done by giving a default argument value of 1 to the parameter times. In the first usage of say, we supply only the string and it prints the string once. In the second usage of say, we supply both the string and an argument 5 stating that we want to say the string 5 times.

Important

Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value before a parameter without a default argument value, in the order of parameters declared, in the function parameter list. This is because 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 you want to specify only some parameters, then you can give values for such parameters by naming them - this is called keyword arguments. We use the name instead of the position which we have been using all along. This has two advantages - One, using the function is easier since we do not need 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.

Using Keyword Arguments

Example 7.6. Using Keyword Arguments

```
#!/usr/bin/python
# Filename : func_key.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 func_key.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 default argument values 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 5 and c gets the default value of 10.

In the second usage `func(25, c=24)`, the variable `a` gets the value of 25 due to the position the argument. Then, the parameter `c` gets the value of `24` due to naming i.e. keyword arguments. The variable `b` gets the default value of `5`.

In the third usage `func(c=50, a=100)`, we use keyword arguments to specify the values. Notice that we are giving values to parameter `c` before `a` here even though `a` is defined before `c` in the function definition list.

The return statement

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

Using the return statement

Example 7.7. Using the `return` statement

```
#!/usr/bin/python
# Filename : return.py

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

print max(2, 3)
```

Output

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

How It Works

The `max` function returns the maximum of the parameters i.e. numbers supplied to the function. When it determines the maximum, it returns that value.

Note that a `return` statement without a value is equivalent to `return None`. `None` is a special value in Python which presents nothingness. For example, it is used to indicate that a variable has no value if the variable has a value of `None`.

Every function implicitly contains a return None statement. You can see this by running print someFunction() where the function someFunction does not use the return statement such as

```
def someFunction():
    pass
```

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

DocStrings

Python has a nifty feature called documentation strings which are usually referred to by their shorter name docstrings. DocStrings are an important tool that you should make use of since it helps to document the program better. We can even get back the docstring from a function at runtime i.e. when the program is running.

Using DocStrings

Example 7.8. Using DocStrings

```
#!/usr/bin/python
# Filename : func_doc.py

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

    The two values must be integers. If they are floating
    point numbers, then they are converted to integers.'''

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

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

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

Output

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

    The two values must be integers. If they are floating
```

```
    point numbers, then they are converted to integers.
```

How It Works

A string on the first logical line of a function is a docstring for that function. 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. Then the second line is blank followed by any detailed explanation starting from the third line. You are strongly advised to follow such a convention for all your docstrings for all your functions.

We access the docstring of the printMax function using the __doc__ attribute of that function. Just remember that Python treats everything as an object including functions. Objects will be explored in detail in the chapter on object-oriented programming.
If you have used the help() in Python, then you have already seen the usage of docstrings! What it does is just fetch the __doc__ attribute of the function and prints it for you. You can try it out on the function above. Just include the help(printMax) statement.

Remember to press q to exit the help().

Automated tools can retrieve documentation from your program in this manner. Therefore, I strongly recommend that you use docstrings for any nontrivial function that you write. The pydoc command that comes with your Python distribution works similarly to help() using docstrings.

Summary

We have seen many aspects of functions. Note that we still have not covered all aspects but this is more than enough to handle any situation using functions.
Next, we will see how to use and create modules.

Chapter 8. Modules

Table of Contents

Introduction

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules. A module is basically a file containing all your functions and variables that you have defined. The filename of the module must have a .py extension.

A module can be imported by another program to make use of its functionality. This is how we use the Python standard library as well. First, we will see how to use the standard library modules.

Using the sys module

Example 8.1. Using the sys module

```
#!/usr/bin/python
# Filename : using_sys.py

import sys

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

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

Output

```
$ python using_sys.py we are arguments
The command line arguments used are:
using_sys.py
we
are
arguments


The PYTHONPATH is ['', '/usr/lib/python2.2',
'/usr/lib/python2.2/plat-linux2', '/usr/lib/python2.2/lib-tk',
'/usr/lib/python2.2/lib-dynload', '/usr/lib/python2.2/site-packages',
'/usr/lib/python2.2/site-packages/gtk-2.0']
```

How It Works

First, we import a module using the import statement. Here, we import the sys module which contains some functionality related to the Python interpreter and its environment. When Python comes to the import sys statement, it looks for the file sys.py in one of the directories listed in the sys.path variable. If the file is found, then the statements in the main block of that module is run, and then the module is made available for you to use. Note that the initialization is done only the first time that we import a module. Also, "sys" is short for "system".

The argv variable in the sys module is referred to using the notation sys.argv. One of the advantages of this approach is that it does not clash with any argv variable declared in our program. Also, it indicates that this variable belongs to the sys module and has not been defined in our module.

The sys.argv variable is a list of strings. We will learn more about lists in later chapters. What you need to know right now is that a list contains an ordered collection of items. The sys.argv variable contains the list of command line arguments i.e. arguments passed to the your program using the command line. Be especially careful to pass command line arguments as shown in the above output if you are using an IDE.

In this case, when we execute python using_sys.py we are arguments, we are executing the program using_sys.py with the python command. The other things are the arguments stored in the sys.argv variable. Remember, the name of the script running is always the first argument in the sys.argv. 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].

We then use the for..in loop to iterate over this list and we print each argument.

The sys.path contains the list of directory names where modules are imported from. Observe that the first string in sys.path is empty - this empty string indicates the current directory which is also part of the sys.path (this is same as the PYTHONPATH environment variable). This means that you can directly import modules located in the

current directory. Otherwise, you will have to place your module in one of the directories listed in sys.path.

Byte-compiled .pyc files

Importing a module is a relatively costly affair, so Python does some optimizations to create byte-compiled files with the extension .pyc . If you import a module such as, say, module.py, then Python creates a corresponding byte-compiled module.pyc . This file is useful when you import the module the next time (even from a different program) - it will be much faster. These byte-compiled files are platform-independent.

The from..import statement

If you want to directly import the argv variable into your program, then you can use the from sys import argv statement. If you want to import all the functions, classes and variables in the sys module, then you can use the from sys import * statement. This works for any module. In general, avoid using the from..import statement and use the import statement instead since your program will be much more readable that way.

A module's __name__

Every module has a name and statements in a module can find out this name. This is especially handy in one particular situation. As mentioned previously, when a module is imported, the main block in that module is run. What if we want to run the block only if the program was used by itself and not when it was imported as a module? This can be achieved using the __name__ variable.

Using a module's __name__

Example 8.2. Using a module's __name__

```
#!/usr/bin/python
# Filename : using__name__.py

if __name__ == '__main__':
        print 'I am here only if this program is run by itself'
        print 'and not imported as a module'
```

Output

```
$ python using__name__.py
This is run only if this program is run by itself
and not imported as a module
```

```
$ python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import using__name__
>>>
```

How It Works

Every Python module/program has a variable defined called __name__ which is set to '__main__' when the program is run by itself. If it is imported to another program, it is set to the name of the module. We make use of this to run a block of statements only if the program was run by itself.

Making your own Modules

Creating your own modules is easy, you have been doing it all along! Every Python program is also a module. The following example should make it clear.

Creating your own Modules

Example 8.3. How to create your own module

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

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

version = '0.1'

# End of mymodule.py
```

The above was a sample 'module'. As you can see, there is nothing special in it compared to our usual Python programs. The following is a Python program that uses this module. Remember that the module should be placed in the same directory as the program or in one of the directories listed in sys.path.

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

import mymodule
```

```
mymodule.sayhi()
print mymodule.version
```

Output

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

from..import

Here is also a version utilising the from..import syntax.

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

from mymodule import *
# You can also use:
# from mymodule import sayhi, version

sayhi()
print version
```

The output of mymodule_demo2.py program is the same as the output of mymodule_demo.py program.

The dir() function

You can use the built-in dir function to list the identifiers that a module defines. The identifiers are the functions, classes and variables defined. When you supply a module name to the dir() function, it returns the list of the names defined in that module. When no argument is supplied to it, it returns the list of names defined in the current module.

Using the dir function

Example 8.4. Using the dir function

```
$ python
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__stderr__', '__stdin__', '__stdout__', '_getframe', 'argv',
'builtin_module_names', 'byteorder', 'copyright', 'displayhook',
```

```
'exc_info', 'exc_type', 'excepthook', 'exec_prefix', 'executable',
'exit', 'getdefaultencoding', 'getdlopenflags', 'getrecursionlimit',
'getrefcount', 'hexversion', 'maxint', 'maxunicode', 'modules',
'path', 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval',
'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace',
'stderr', 'stdin', 'stdout', 'version', 'version_info', 'warnoptions']

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

How It Works

Notice how the assigning of a variable automatically adds that identifier name to the list returned by the dir() function. When we delete the variable i.e. undeclare the variable, then it is automatically removed from the list returned by the dir() function. We delete a variable using the del statement. After the statement del a, you can no longer access the variable a (unless you define it again, of course) - it was as if it never existed in the current program.

Summary

We have seen how to use modules and create our own modules. Modules are useful because they provide services and functionality that you can reuse in your programs. The standard library is an example of a set of such modules.
Next, we will see some more interesting stuff called data structures.

Chapter 9. Data Structures

Table of Contents

Introduction

Data structures are basically just that - they are structures which hold data together. They are used to store a collection of related data. There are three built-in data structures in Python - list, tuple and dictionary.

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 can think of a shopping list where you have a list of items you want to buy, except that you probably have each item on 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. You can add, remove or search for items in a list.

Objects and Classes

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

A list is an example of usage of objects and classes. When you use a variable i and assign an integer, say 5 to it, 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 also have methods i.e. functions defined for use with respect to that class only i.e. you can use these pieces of functionality only when you have an object of that class. For example, Python provides an append method for the list class which allows you to add an item to the list. For example, mylist.append('an item') will add that string to the list mylist. Note the use of dot notation for accessing methods of objects.

A class can also have fields which are nothing but variables defined for use with respect to that class only i.e. you can use these variables only when you have an object of that class. Fields are also accessed by dot notation, such as mylist.field.

Using Lists

Example 9.1. Using Lists

```
#!/usr/bin/python
# Filename : list.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
for item in shoplist:
        print item,

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

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 now is', shoplist
```

Output

```
$ python list.py
```

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

How It Works

The variable shoplist is a shopping list for someone who is going to the market. Here, I am storing just strings in the list but remember that you can add anything to the list i.e. you can add any object to the list - even numbers or other lists.

We have also used the for..in loop to go through the items of the list. By now, you should have realised that a list is also an example of a sequence. The speciality of sequences will be discussed in detail later.

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 a bit of an ugly way of doing it, but it gets the job done.

Next, we add an item to the list using the append method of the list object, as discussed before. Then, we check that the item has been indeed added to the list by printing the contents of the list. Note that the print statement automatically prints the list in a neat manner for us.

Then, we sort the list by using the sort method of the list object. Remember that this method affects the list itself and does not return a changed list - this is different from strings. This is what we mean by saying that lists are mutable and that strings are immutable.

Next, when we finish buying an item in the market, we want to remove it from the list. We achieve this using the del statement. Here, we mention which item of the list we want to remove and the del statement removes it from the list for us. Then, we just print the list to check that it has been indeed removed from the list.

We can access members of the list by using their position as shown above. Remember that Python starts counting from 0. Therefore, if you want to access the first item in a list then you can use mylist[0] to get the first item in the list.
If you want to know all the methods defined by the list object, see help(list) for complete details.

Tuple

Tuples are just like lists except that they are immutable (like strings) 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 user-defined function can safely assume that the collection of values i.e. the tuple of values used will not change.

Using Tuples

Example 9.2. Using Tuples

```
#!/usr/bin/python
# Filename : tuple.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 new_zoo       # Prints all the animals in the new zoo
print new_zoo[2]    # Prints animals brought from the old zoo
print new_zoo[2][2] # Prints the last animal brought from the old zoo
```

Output

```
$ python tuple.py
Number of animals in the zoo is 3
Number of animals in the new zoo is 3
('monkey', 'dolphin', ('wolf', 'elephant', 'penguin'))
('wolf', 'elephant', 'penguin')
penguin
```

How It Works

Here, the variable zoo refers to a tuple of items. We see that the len function can be used to get the length of a tuple as well. 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 which are already there along with the animals brought over from the old zoo. Back to reality, note that a tuple within a tuple does not lose it's identity.

We can access the items in the tuple using the indexing operator just like we did for lists. We just specify the item's position within a pair of square brackets following the name of the tuple such as new_zoo[2]. In this case, this object is a tuple. So we can access items of this object as well using new_zoo[2][2].

Tuple with 0 or 1 items. An empty tuple is constructed by an empty pair of parentheses such as myempty = (). However, a tuple with a single item is not so simple. You have to specify it using a comma following the single item so that Python can differentiate between a tuple and a pair of parentheses used for grouping in an expression i.e. you have to specify singleton = (some_item ,) .

> **Note to Perl programmers**
> A list within a list does not lose it's 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. As far as Python is concerned, they are just objects stored using another object, that's all.

Tuples and the print statement

Tuples are most often used alongwith the print statement. An example will make this clear.

Example 9.3. Output using tuples

```
#!/usr/bin/python
# File : tuple.py

age = 21
name = 'Swaroop'

print '%s is %d years old' % (name, age)
print "Why is %s playing with that python?" % name
```

Output

```
$ python tuple.py
Swaroop is 21 years old
Why is Swaroop playing with that python?
```

How It Works

The print statement takes a string using certain specifications followed by a % symbol which is followed by a tuple. The string can have specifications such as %s for strings and %d for integers. The tuple must have items corresponding to these specifications in the same order.

Observe the first usage where we have used %s first and this corresponds to the string name which is the first item in the tuple. Then, the second specification is %d which corresponds to the number age which is the second item in the tuple.

What Python does here is that it converts each item in the tuple into a string and substitutes that string value into the place of the specification. Therefore the %s in the first usage will be replaced by the value of the name variable, and so on.

This usage of the print statement makes writing output extremely easy and avoids using commas everywhere as we have done until now.

Most of the time, you can just use the %s specification and let Python take care of the rest for you. This works even for numbers, but you may want to give correct specifications in order to ensure that objects of proper type are being used.
In the second usage, we are using a single specification in the string followed by the % symbol followed by a single item - there are no pair of parentheses. 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 values (details). Note that the key must be unique - you cannot find out the correct information if you have two persons with the exact same name.

A word of caution: you can use only immutable values (like strings) for keys of a dictionary but you can use either immutable or mutable values for values. This basically means to say that you can use only simple objects as keys.

Pairs of keys and values are specified in a dictionary by using the notation d = {key1 : value1, key2 : value2 }. Notice that the key and value pairs are separated by a colon, and the pairs are separated themselves by commas and all this is enclosed in a pair of curly brackets.

Remember that key/value pairs in a dictionary are not ordered in any manner. If you want a particular order, then you will have to sort them yourself.
The dictionaries that you will be using are objects of class dict.

Using Dictionaries

Example 9.4. Using dictionaries

```
#!/usr/bin/python
# Filename : dict.py

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

```
ab = {        'Swaroop' : 'python@g2swaroop.net',
              'Miguel'  : 'miguel@novell.com',
              'Larry'   : 'larry@wall.org',
              'Spammer' : 'spammer@hotmail.com'
      }

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

# Adding a key/value pair
ab['Guido'] = 'guido@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 ab.has_key('Guido'):
      print "\nGuido's address is %s" % ab['Guido']
```

Output

```
$ python dict.py
Swaroop's address is python@g2swaroop.net

There are 4 contacts in the address-book

Contact Swaroop at python@g2swaroop.net
Contact Larry at larry@wall.org
Contact Miguel at miguel@novell.com
Contact Guido at guido@python.org

Guido's address is guido@python.org
```

How It Works

We create the dictionary `ab` using the notation we have already discussed. We then access key/value pairs by specifying the key using the indexing operator as discussed in the context of lists and tuples.

We can also add new key/value pairs using the indexing operator - we just access that key and assign that value, as we have done for Guido in the above case.

We can delete key/value pairs using our old friend - the del statement. We specify which key/value pair by specifying the dictionary followed by an index operation for that key with the del statement (there is no need to use the value for this operation).

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 two items - the key followed by the value. We retrieve these two items in each tuple and assign them to the variables name and address. Then, we use the for..in loop to iterate over this and we print these values in the for-block.

We can check if a key/value pair exists using the has_key method of a dictionary as used above. You can see documentation for the complete list of methods of the dict class using help(dict).

Keyword Arguments and Dictionaries. On a different note, if you have used keyword arguments in your functions, you have already used dictionaries! Just think about it - the key/value pair is specified by you in the parameter list of the function definition and when you access variables within your function, it is just a key access of a dictionary (which is called the symbol table in compiler design terminology).

Sequences

Lists, tuples and strings are examples of sequences, but what is so special about sequences? Two of the main features of a sequence is the indexing operation which allows us to fetch a particular item in the sequence and the slicing operation which allows us to retrieve a slice of the sequence i.e. a part of the sequence.

Using Sequences

Example 9.5. Using sequences

```
#!/usr/bin/python
# Filename : seq.py

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

# Indexing or 'Subscription'
print shoplist[0]
print shoplist[1]
print shoplist[2]
print shoplist[3]
print shoplist[-1]
print shoplist[-2]

# Slicing using a list
print shoplist[1:3]
print shoplist[2:]
```

```
print shoplist[1:-1]
print shoplist[:]

# Slicing using a string
name = 'swaroop'
print name[1:3]
print name[2:]
print name[1:-1]
print name[:]
```

Output

```
apple
mango
carrot
banana
banana
carrot
['mango', 'carrot']
['carrot', 'banana']
['mango', 'carrot']
['apple', 'mango', 'carrot', 'banana']
wa
aroop
waroo
swaroop
```

How It Works

First, we see how to use indexes to get individual items of a sequence. This is also referred to as subscription. Whenever you specify a number to a sequence within square brackets as shown above, Python will fetch you the item corresponding to that position in the sequence. Remember that Python starts counting numbers 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, shoplist[-1] fetches the last item and shoplist[-2] fetches the second last item in the shoplist sequence.

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 you have been using till now. Remember the numbers are optional when using slices but the colon isn't.

The first number is the position from where the slice starts and the second number is where the slice will stop at. If the first number is left out, Python defaults to

the beginning of the sequence. If the second number is left out, Python defaults to the end of the sequence.

Thus, shoplist[1:3] returns a slice of the sequence starting at position 1, includes position 2 but stops at position 3 i.e. it does not include position 3. Therefore a slice of 2 items is returned. Also, shoplist[:] returns a copy of the whole sequence.

You can also try out slicing with negative numbers. Negative numbers are used for positions 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.

Try various combinations of such slice specifications using the Python interpreter interactively i.e. the prompt so that you can see the results quickly. The great thing about sequences is that you can access tuples, lists and strings all in the same way.

References

Lists are examples of objects. When you create an object and assign it to a variable, the variable only refers to the object and is not the object itself i.e. the variable points to that part of your computer's memory where the list is stored. Generally, you don't 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.

Objects and References

Example 9.6. Objects and references

```
#!/usr/bin/python
# Filename : reference.py

shoplist = ['apple', 'mango', 'carrot', 'banana']
mylist = shoplist
# mylist is just another reference to the same list!

del shoplist[0]
# I purchased the first item, so I remove it from the list.

print 'shoplist is', shoplist
print 'mylist is', mylist
# Notice that shoplist and mylist both print a list without the
# 'apple' confirming that they refer to the same list object

mylist = shoplist[:]
# Obtain a full slice to make a copy
del mylist[0]

print 'shoplist is', shoplist
print 'mylist is', mylist
```

```
# Notice now that the two lists are different
```

Output

```
$ python reference.py
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

How It Works

Most of the explanation is available in the comments itself. What you need to remember is that if you want to make a copy of a list or such sequences and objects (not simple objects such as integers), then you have to use a slicing operation without numbers to make a copy. If you just assign the variable name to another variable name, both of them refer to the same object and not different objects.

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

More about Strings

We have already discussed strings in detail earlier. What more can there be to add? Well, did you know that strings are also objects and have methods which do everything from checking substrings to stripping spaces!

The strings that you use in your program are all objects (instances) of the class `str`. Some useful methods of this class are demonstrated in the following example. For a complete list of such methods, see `help(str)`.

String Methods

Example 9.7. String methods

```
#!/usr/bin/python
# Filename : str_methods.py
```

```
name = 'Swaroop' # This is a string object

if name.startswith('Swa'):
        print 'Yes, the string starts with "Swa"'

if 'a' in name:
        print 'Yes, it contains the string "a"'

if name.find('war') != -1:
        print 'Yes, it contains the string "war"'

delimiter = '-*-'
mylist = ['India', 'China', 'Finland', 'Brazil']
print delimiter.join(mylist)
```

Output

```
$ python str_methods.py
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
India-*-China-*-Finland-*-Brazil
```

How It Works

Here, we see a lot of the string methods in action. The `startswith` method is used to find out whether the string starts with the given string. The `in` operator is used to check if a given string is a substring of the string i.e. is part of the string.

The `find` method is used to do the same thing but it returns `-1` when it is unsuccessful and returns the position of the substring when it is successful. The string object also has a neat method called `join` which is used to put the items of a sequence in a string separated by that string.

Summary

We have explored the various built-in data structures of Python in detail. These data structures will be essential for writing programs of reasonable size.
Now that we have a lot of the basics of Python in place, we will see how to design and write a real-world Python program next.

Chapter 10. Problem Solving - Writing a Python Script

Table of Contents

We have explored various parts of the Python language and now we will take a look at how all these parts fit together, by designing and writing a program which does something useful.

The Problem

The problem 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 and will work. In this case, I have created the following list.

1. The files and directories to be backed up are given 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 zip command available by default in any standard Linux/Unix distribution. Windows users can use the Info-Zip program. Note that you can use any archiving command you want 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.

First Version

Example 10.1. Backup Script - The First Version

```
#!/usr/bin/python
# Filename : backup_version1.py

import os
import time

# 1. The files and directories to be backed up are given in a list.
source = ['/home/g2swaroop/all', '/home/g2swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work']

# 2. The backup must be stored in a main backup directory.
target_dir = '/mnt/d/backup/'

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

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

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

Output

```
$ python backup_version1.py
Successful backup to /mnt/d/backup/20031124174143.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 how we have converted our design into code in a step-by-step manner.

We first import the os and time modules to use some of the functionality of these modules. Then, we specify the files and directories to backup in the source list. The target directory is where we store all our backup files and this is specified by the target_dir variable. The name of the zip archive backup that we are going to create is the current date and time as returned by the time.strftime() function with the .zip extension and this archive is stored in the target_dir directory.

The time.strftime() function takes a specification like 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 for the current date, and so on. The complete list of such specifications can be found in the [Python Reference Manual] that comes with your Python distribution.

Then we create the name of the target zip file using the addition operator which concatenates the strings i.e. returns a string which combines those two strings. Then, we create a string zip_command which contains the command that we are going to execute. You can execute this command directly from the shell (Linux terminal or DOS prompt) to check if it works properly.

The zip command that we are using is like this - we use the option -q to indicate that the zip command should work quietly. The option -r indicates that the zip command should work recursively for directories i.e. it should include subdirectories and files within the subdirectories as well. The two options are combined to get -qr. The options are followed by the name of the zip archive to create, followed by the list of files and directories to backup. We convert the source list into a string using the join method of strings which we have already seen how to use.

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. the shell. It then returns 0 if the command was successfully. It will return an error number otherwise.

Depending on the outcome of the command we print an appropriate message and that's it, we have created 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 in Windows, but you have to be a little careful. 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 itself as an escape sequence or you have to use raw strings. For example, use 'C:\\Documents' or use r'C:\Documents', but do not use 'C:\Documents' - you are using an unknown escape sequence \D in this case!

Now that we have a working backup script, we can use it whenever we want to take the backup of files. Linux/Unix users are advised to use the executable method we discussed earlier so that they can run the backup script anytime anywhere. This is called the operation phase or the deployment phase of the software.

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

Second Version

The first version at our script is good, but we can make some refinements to it so that it can work better. This is called the maintenance phase of the software.

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

Example 10.2. Backup Script - The Second Version

```
#!/usr/bin/python
# Filename : backup_version2.py

import os
import time

# The files and directories to backup
source = ['/home/g2swaroop/all', '/home/g2swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work']
```

```
# The directory where to store the backup
target_dir = '/mnt/d/backup/'

# The date - the subdirectory in the main backup directory
today = target_dir + time.strftime('%Y%m%d')
# The time - the name of the zip archive
now = time.strftime('%H%M%S')

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

# The name of the zip file
target = today + os.sep + now + '.zip'

# The zip command to run
zip_command = 'zip -qr %s %s' % (target, ' '.join(source))

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

Output

```
$ python backup_version2.py
Successfully created directory /mnt/d/backup/20031124
Successful backup to /mnt/d/backup/20031124/174239.zip
$ python backup_version2.py
Successful backup to /mnt/d/backup/20031124/174241.zip
```

How It Works

Most of the program remains the same. The addition is that we check if the directory with the current date as name exists inside the main backup directory using the os.exists function. If not, we create it using the os.mkdir function (which is short for make directory). Notice the use of the os.sep variable - this gives the directory separator according to your operating system i.e. it is '/' in Linux/Unix, it is '\\' in Windows and ':' in Mac OS. Using os.sep instead of these characters makes our programs portable.

Third Version

The second version works fine, but when I do many backups, I am finding it hard to differentiate what the backups were for. For example, I might have made some major

changes to a document, then I want to associate what those changes are with the name of the backup archive. This can be achieved by attaching a user-supplied comment to the name of the zip archive.

Example 10.3. Backup Script - The Third Version (does not work!)

```
#!/usr/bin/python
# Filename : backup_version3.py

import os, time

# The files and directories to backup
source = ['/home/g2swaroop/all', '/home/g2swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work']

# The directory where to store the backup
target_dir = '/mnt/d/backup/'

# The date - the subdirectory in the main backup directory
today = target_dir + time.strftime('%Y%m%d')
# The time - the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user
comment = raw_input('Enter a comment --> ')
if len(comment) == 0:    # Check if a comment was entered
        # The name of the zip file
        target = today + os.sep + now + '.zip'
else:
        target = today + os.sep + now + '_' +
                comment.replace(' ', '_') + '.zip'

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

# The zip command to run
zip_command = 'zip -qr %s %s' % (target, ' '.join(source))

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

Output

```
$ python backup_version3.py
  File "backup_version3.py", line 23
    target = today + os.sep + now + '_' +
                                    ^
SyntaxError: invalid syntax
```

How It (does not) Work

This program does not work! Python says there is a syntax error which means that the script does not satisfy the structure that Python it expects. When we observe the error given by Python, we see that it gives us the place where it detected the error as well. So we start debugging our program from that line.

On careful observation, we see that the single logical line has been split into two physical lines and we have not specified that these two physical lines belong together. Basically, Python has found the addition operator (+) without any operand in that logical line. We can specify that the logical line continues in the next physical line by the use of a backslash at the end of the physical line as we have already seen. So we make this correction to our program.

Fourth Version

Example 10.4. Backup Script - The Fourth Version

```
#!/usr/bin/python
# Filename : backup_version4.py

import os, time

# The files and directories to backup
source = ['/home/g2swaroop/all', '/home/g2swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work']

# The directory where to store the backup
target_dir = '/mnt/d/backup/'

# The date - the subdirectory in the main backup directory
today = target_dir + time.strftime('%Y%m%d')
# The time - the name of the zip archive
now = time.strftime('%H%M%S')

# Take a comment from the user
comment = raw_input('Enter a comment --> ')
if len(comment) == 0:    # Check if a comment was entered
        # The name of the zip file
        target = today + os.sep + now + '.zip'
```

```
else:
        target = today + os.sep + now + '_' + \ # Notice backslash
                comment.replace(' ', '_') + '.zip'

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

# The zip command to run
zip_command = 'zip -qr %s %s' % (target, ' '.join(source))

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

Output

```
$ python backup_version4.py
Enter a comment --> fixed bug
Successful backup to /mnt/d/backup/20031124/181157_fixed_bug.zip
$ python backup_version4.py
Enter a comment -->
Successful backup to /mnt/d/backup/20031124/181202.zip
```

How It Works

This program now works. Let us go through the actual enhancements that we had made in version 3. We take the user's comment using the raw_input function and then check if the user actually entered something or not. If the user has just pressed enter for some reason (maybe it was a routine backup and no special changes were made), then we proceed as before.

However, if a comment was supplied, then this is attached to the name of the zip archive just before the .zip extension. Notice that we replace spaces in the comment with underscores because managing such filenames are easier.

More Refinements

The fourth version must be 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 -v option to make your program more talkative, or you can backup additional files and directories specified on the command line using the sys.argv list.

One refinement I prefer is the use of the tar command instead of the zip command in Linux/Unix. One advantage is that when you use tar along with gzip, the backup is much faster and the archive size created is also much smaller. If I need to use this archive in Windows, then WinZip handles such .tar.gz files as well.

The command to use for utilising the tar is

```
tar = 'tar -cvzf %s %s -X /home/g2swaroop/bin/excludes.txt' % (dst,
      ' '.join(srcdir))
```

where the options 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. over-writing.
- -X indicates a file which contains a list of filenames which must be included from backup. For example, you can specify *~ in this file to not include any filenames ending with ~ in the backup.

Note
An even better way of creating a backup script is to use the zipfile module included in the Python Standard Library. This avoids using os.system() which is generally not advisable to use.
For pedagogical purposes, I decided to use os.system() so that the example is simple enough to be understood by everybody but real enough to be useful.

The Software Development Process

We have now gone through the 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. Use (Operation or Deployment)

6. Maintain (Refinement)

Important
A recommended way of writing programs is the procedure we have followed here - Do the analysis and design. Start implementing with a simple version. Test and debug it. Use it to ensure that it behaves as expected. Now, add any features you want and continue to repeat the Do It-Test-Use cycle as many times as required. Remember, "Software is grown, not built" .

Summary

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

Next, we will discuss object-oriented programming.

Chapter 11. Object-Oriented Programming

Table of Contents

Introduction

In our programs till now, we designed our program around functions or blocks of statements which manipulate data. This is called the procedural programming paradigm. There is another way of organising your program which is to combine data and functionality and wrap it inside what is called an object. This is called the object oriented programming paradigm. Most of the time you can use procedural programming but when you want to write large programs or if you have a program that is 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 can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class.

Note to C/C++/Java/C# Programmers

Please note that even integers are treated as objects of the class int. This is unlike C++ and Java (<1.5) where integers are magic types. See help(int) for more details.

This level of object-orientation in Python is strikingly different from other languages, especially, when compared to C++ and Perl. However, C# programmers will be familiar with this concept since it closely resembles the boxing and unboxing technique (see the Microsoft C# and Mono [go-mono.com] websites). Now, even Java 1.5 has the same thing which it refers to as autoboxing and auto-unboxing (see the Java website).

Objects can store data using ordinary variables that belong to the object. Variables that belong to an object or class are called as fields. Objects can also have functionality by using functions that belong to the class. Such functions are called methods. This terminology is important because it helps us to differentiate between a function which is separate by itself and a method which belongs to an object.

Remember, that fields are of two types - they can belong to each instance (object) of the class or they 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 an indented block.

The self

Class methods have only one specific difference from ordinary functions - they have an extra variable that has to be added to the beginning of the parameter list, but you do not give a value for this parameter when you call the method. This particular variable refers to the object itself, and by convention, it is given the name 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 that it is the object variable i.e. the self and even specialized IDEs (Integrated Development Environments such as Boa Constructor) can help you if you use this particular name.

> **Note to C++/Java Programmers**
> The self variable is equivalent to the this pointer in C++ and the this reference in Java.

You must be wondering why you don't need to give a value for this parameter just like you do for other parameters. The reason is that Python will automatically provide this value. For example, if you have a class called MyClass and an instance (object) of this class called MyObject, then when you call a method of this object as MyObject.method(arg1, arg2), this is automatically converted to MyClass.method(MyObject, arg1, arg2). This is what the special self is all about.

Classes

The simplest class possible is shown in the following example.

Creating a Class

Example 11.1. Simplest Class

```
#!/usr/bin/python
# Filename : simplestclass.py

class Person:
        pass # A new block

p = Person()
print p
```

Output

```
$ python simplestclass.py
<__main__.Person instance at 0x816a6cc>
```

How It Works

We create a new class using the class statement followed by the name of the class (Person in this case),followed by a block of statements with a higher level of indentation which forms the body of the class. In this case, we have an empty block which is indicated 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 discuss instantiation of objects in more detail later. For our verification, we confirm the type of the variable i.e. object using the print statement. Notice that the address and the type of the object is printed. The address will have a different value on your computer, but the type confirms that we have indeed created an object of the class Person.

Object Methods

We have already discussed that classes/objects can have methods which are just like functions except for the usage of the self variable. Even if your method does not take any parameters, you still have to have the self variable.

Object Methods

Example 11.2. Using Object Methods

```
#!/usr/bin/python
# Filename: methods.py
```

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

p = Person()
p.sayHi() # This short example can also be written as Person().sayHi()
```

Output

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

How It Works

Here we see the self variable in action. Notice, that the sayHi method takes no parameters but still has the self variable in it's parameter list in the function definition. Other than this, methods are no different from functions.

The __init__ method

You can define a special method for a class with the name __init__ which is run as soon as an object of this class is instantiated. This method is used for any initialization you want to do with your object. The next example will demonstrate this.

> **Note to C++/Java/C# Programmers**
> The __init__ method is analogous to a constructor in C++ or C# or Java.

Class and Object Variables

We will now discuss the data part of the object - the variables. We have two types - the class variables and the object variables. The difference is in the ownership - does the class own the variables or does the object own the variables?

When the class owns the variables, it is called a class variable. Class variables are shared in the sense that they can be accessed by all objects (instances) of that class. When the object owns the variables, it is called an object variable. In this case, each object has its own copy of this variable i.e. they are not shared and are not related in any way to the variable of the same name in a different instance of the same class. An example will make this clearer.

Using Class and Object Variables

Example 11.3. Using Class and Object Variables

```
#!/usr/bin/python
# Filename: objvar.py

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

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

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

	def sayHi(self):
		'''Greets the other person.

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

	def howMany(self):
		'''Prints the current population.'''
		# There will always be atleast one person
		if Person.population == 1:
			print 'I am the only person here.'
		else:
			print 'We have %s persons here.' % \
			Person.population


swaroop = Person('Swaroop')
swaroop.sayHi()
swaroop.howMany()

kalam   = Person('Abdul Kalam')
kalam.sayHi()
kalam.howMany()

swaroop.sayHi()
swaroop.howMany()
```

Output

```
$ python objvar.py
(Initializing Swaroop)
Hi, my name is Swaroop.
I am the only 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.
```

How It Works

This example, although a long one, helps demonstrate the nature of class and object variables. Here, we refer to the population variable of the Person class as Person.population and not as self.Population. Note that an object variable with the same name as a class variable will hide the class variable!

We refer to the object variable name using self.name in the methods. Remember this simple difference between class and object variables. To summarize, ClassName.field1 refers to the class variable called field1 of the class ClassName. This variable is shared by all instances/objects of this class. The variable self.field2 refers to the object variable of the class and is different in different objects.

Observe that the __init__ method is run first even before we get to use the object and this method is used to initialize the object variables for later use. This is confirmed by the output of the program which indicates when the particular object is initialized.

Notice that when we change the value of Person.population, all the objects use the new value which confirms that the class variables are indeed shared by all the instances of that class. We can also observe that the values of the self.name variable is specific to each object which indicates the nature of object variables. Remember, that you must refer to the variables and methods of the same object using the self variable only. This is called an attribute reference.

In this program, we can also see the use of docstrings for classes as well as methods. We can access the class docstring at runtime such as Person.__doc__ and the method docstring as Person.sayHi.__doc__.

Just like the __init__ method, we can also have a __del__ method where we can decrement the Person.population by 1. This method is run when the object is no longer in use. If you want to try this out, add the __del__ method and then use the statement del personInstance to delete the object. You can use a print statement within this method to see when it is run.

Note to C++/Java/C# Programmers

All class members (including the data members) are public and all the methods are virtual in Python.

One exception: If you use data members with names using the double underscore prefix such as __privatevar, Python uses name-mangling to effectively make it a private variable.

Note to C++ Programmers

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

Inheritance

One of the major benefits of object oriented programming is reuse of code and one of the ways this is achieved 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 your college. They have some common characteristics such as name, age and address. They also have some specific characteristics. For example, teachers have salary, courses and leaves and students have marks and fees.

Although you could have created two independent classes for each type, a better way would be to create a common class called SchoolMember and then have the teacher and student classes inherit from this class i.e. be sub-types of this type (class) and adding the specific characteristics or functionality to the sub-type.

There are many advantages to this approach. One is that you can refer to a teacher or student object as a SchoolMember object, which could be helpful in some situations such as counting the number of school members. This is called polymorphism where a sub-type can be substituted in any situation where a parent type is expected i.e. the object can be treated as an instance of a parent class.

Another advantage is that changes to the SchoolMember class are reflected in the teacher and student classes as well. For example, you can add a new identification number for each school member i.e. both teachers and students using this mechanism. More importantly, we reuse the code of the parent class and we do not need to repeat it elsewhere such as in the sub-types.

The SchoolMember class in this situation is known as the base class or the superclass. The Teacher and Student classes are called the derived classes or subclasses.

Inheritance

Example 11.4. Inheritance

```
#!/usr/bin/python
```

```
# Filename: inheritance.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 tell(self):
        print '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 tell(self):
        SchoolMember.tell(self)
        print 'Salary:"%d"' % 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 tell(self):
        SchoolMember.tell(self)
        print 'Marks:"%d"' % self.marks


t = Teacher('Mrs. Abraham', 40, 30000)
s = Student('Swaroop', 21, 75)

print # prints a blank line

members = [t, s]
for member in members:
    member.tell()
    # Works for instances of Student as well as Teacher
```

Output

```
$ python inheritance.py
(Initialized SchoolMember: Mrs. Abraham)
(Initialized Teacher: Mrs. Abraham)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Abraham" Age:"40"  Salary:"30000"
Name:"Swaroop" Age:"21"  Marks:"75"
```

How It Works

To use inheritance, we specify the base class names in a parenthesized list following the class name in the class definition. Next, we can observe that the __init__ method of the base class is explicitly called using the self variable so as to initialize the base class part of the object. We can also observe that we can treat instances of Teacher or Student as just instances of SchoolMember when we use the tell method of the SchoolMember class.

A note on terminology - if more than one class is listed in the inheritance list, then it is called multiple inheritance.

Summary

We have now explored the various aspects of classes and objects as well as the various terminologies associated with it. We have also seen the various benefits and pitfalls of object-oriented programming. Python is highly object-oriented and it may be beneficial to understand these concepts thoroughly.

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

Chapter 12. Input/Output

Table of Contents

There will be lots of instances where your program needs to interact with the user (which could be yourself) and we have already seen how to do this with the help of the raw_input function and the print statement. You can also the various string methods i.e. methods of the str class. For example, you can use the rjust method of the str class to get a string which is right justified to a specified width. See help(str) for more details.

Another common type of input/output you need to do is with respect to 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 first creating an object of the file class. Then we use the read, readline, or write methods of the file object to read from or write to the file depending on which mode you opened the file in. Then finally, when you are finished the file, you call the close method of the file object.

Using file

Example 12.1. Using files

```
#!/usr/bin/python
# Filename: fileob.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')
f.write(poem)
f.close()

f = file('poem.txt') # the file is opened in 'r'ead mode by default
while True:
        line = f.readline()
```

```
    if len(line) == 0: # Length 0 indicates EOF
        break
    print line, # So that extra newline is not added
f.close()
```

Output

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

How It Works

First, we create an instance of the file class and specify the name of the file we want to access and the mode in which we want to open the file. The mode can be a read mode('r'), write mode ('w') or the append mode ('a'). There are actually many more modes available and help(file) should give you more details.

In this case, we open the file in write mode. Then, we use the write method of the file object to write to the file. Finally, we call the close method to finish.

Then, we open the same file again for reading. Notice that if we don't specify the mode, then the read mode is the default one. We read each line of the file using the readline method in a loop. This method returns a complete line, including the newline character. So, even an empty line will have a single character which is the newline. The end of the file is indicated by a completely empty string which is checked for using len (line) == 0.

Notice that we use a comma with the print statement to suppress the automatic newline of the print statement because the line that is read from the file already ends with a newline character. Then, we close the file. See the poem.txt file to confirm that the program has indeed worked properly.

Pickle

Python provides a standard module called pickle which you can use to store any Python object to a file and then get it back later. This is called storing the object persistently.

There is another module called cPickle which acts just the pickle module except that is written in the C language and is (upto 1000 times) faster. You can use either of

these modules, although we will be using the cPickle module here. Remember though, that here we refer to both these modules as the pickle module.

Pickling and Unpickling

Example 12.2. Pickling and Unpickling

```
#!/usr/bin/python
# Filename: pickling.py

import cPickle

shoplistfile = 'shoplist.data'  # The name of the file we will use

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

# Write to the storage
f = file(shoplistfile, 'w')
cPickle.dump(shoplist, f) # dump the data to the file
f.close()

del shoplist # Remove shoplist

# Read back from storage
f = file(shoplistfile)
storedlist = cPickle.load(f)
print storedlist
```

Output

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

How It Works

We create a file object in write mode and then store the object into the opened file by calling the dump function of the pickle module which stores the object into the file. This process is called pickling.

Next, we retrieve the object using the load function of the pickle module which returns the object. This process is called unpickling.

Summary

We have discussed various types of input/output including files and using the pickle module. This will help you to manipulate files with ease.

Next, we will explore exceptions.

Chapter 13. Exceptions

Table of Contents

Exceptions occur when certain exceptional situations occur in your program. For example, what if you are reading a file and you accidentally deleted it in another window or some other error occurred? Such situations are handled using exceptions.

What if your program had some invalid statements? This is handled by Python which raises its hands and tells you there is an error.

Errors

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

```
>>> Print 'Hello, World'
  File "<stdin>", line 1
    Print 'Hello, World'
                  ^
SyntaxError: invalid syntax
>>> print 'Hello, World'
Hello, World
>>>
```

Observe that a `SyntaxError` is raised and also the location where the error was detected, is printed. This is what a handler for the error does.

Try..Except

To show the usage of exceptions, we will try to read input from the user and see what happens.

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

```
>>>
```

Here, we ask the user (which is you in this case) for input and if you press Ctrl-d i.e. the EOF (end of file) character, then Python raises an error called EOFError. Next, we will see how to handle such errors.

Handling Exceptions

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

Example 13.1. Handling Exceptions

```
#!/usr/bin/python
# Filename: try_except.py

import sys

try:
    s = raw_input('Enter something --> ')
except EOFError:
    print '\nWhy did you do an EOF on me?'
    sys.exit() # Exit the program
except:
    print '\nSome error/exception occurred.'
    # Here, we are not exiting the program

print 'Done'
```

Output

```
$ python try_except.py
Enter something -->
Why did you do an EOF on me?
$ python try_except.py
Enter something --> Python is exceptional!
Done
```

How It Works

We put all the statements that might raise an error in the try block and then handle all errors and exceptions in the except clause/block. The except clause can handle a single specified error or exception or a parenthesized list of errors/exceptions. If no names of errors or exceptions are supplied, 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 the default Python handler is called which stops the execution of the program and prints a message. We have already seen how this works.

You can also have an else clause with the try..catch block. The else clause is executed if no exception occurs.

We can also get the exception object 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 specify the name of the error/exception and the exception object. The error or exception that you can raise should be a class which directly or indirectly is a derived class of the Error or Exception class respectively.

How To Raise Exceptions

Example 13.2. Raising Exceptions

```
#!/usr/bin/python
# Filename: raising.py

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

try:
    s = raw_input('Enter something --> ')
    if len(s) < 3:
        raise ShortInputException(len(s), 3)
    # Other work can go as usual here.
except EOFError:
    print '\nWhy did you do an EOF on me?'
except ShortInputException, x:
    print '\nThe input was of length %d, it should be at least %d'\
        % (x.length, x.atleast)
else:
    print 'No exception was raised.'
```

Output

```
$ python raising.py
Enter something -->
Why did you do an EOF on me?
$ python raising.py
Enter something --> ab

The input was of length 2, it should be atleast 3
$ python raising.py
Enter something --> abc
No exception was raised.
```

How It Works

Here, we have created our own exception type, although we could've used any predefined exception/error for demonstration purposes. This new exception type is the class ShortInputException. It declares two fields - length and atleast which is the length of the input and the minimum length that the input should have been.

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. Inside this particular except clause, we use the length and atleast fields to print an appropriate message to the user.

Try..Finally

What if you wanted some statements to execute after the try block whether or not an exception was raised? This is done using the finally block. Note that if you are using a finally block, you cannot have any except clauses for the same try block.

Using Finally

Example 13.3. Using Finally

```
#!/usr/bin/python
# Filename: finally.py

try:
    f = file('poem.txt')
    while True: # Our usual file-reading block
        l = f.readline()
        if len(l) == 0:
            break
        print l,
finally:
    print 'Cleaning up...'
    f.close()
```

```
```

Output

```
$ python finally.py
Programming is fun
When the work is done
if (you wanna make your work also fun):
      use Python!
Cleaning up...
```

How It Works

Here, we do the usual file-reading stuff that we have done before, but we ensure that the file is closed even if an IOError or any other error/exception is raised when the file is being opened or read.

Summary

We have discussed the usage of the try..except and the try..finally statements. We have seen how to create our own exception types and how to raise exceptions as well. Next, we will explore the Python Standard Library.

Chapter 14. The Python Standard Library

Table of Contents

Introduction

The Python Standard Library is available with every Python installation. It contains a huge number of very useful modules. It is important that you become familiar with the Python Standard Library since most of your problems can be solved more easily and quickly if you are familiar with this library of modules.

We will now explore the Python standard library and some of the most commonly used modules in this library. The complete documentation for the Python Standard Library is available with the standard documentation that comes with your Python installation. The "Library Reference" section in the Python Documentation will give you the complete details of the modules available.

The sys module

The sys module contains system-specific functionality. We have already seen that the sys.argv list contains the command line arguments. Let us see an example.

Command Line Arguments

Example 14.1. Using sys.argv

```
#!/usr/bin/python
# Filename : cat.py
import sys

##### Functions #####
def readfile(filename):
    '''Print a file to the standard output.'''
    f = file(filename)
    while True:
        line = f.readline()
        if len(line) == 0:
            break
        print line, # The comma is to suppress additional newline.
    f.close()

##### Main #####
```

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

if sys.argv[1].startswith('--'):
    option = sys.argv[1][2:]
    # Fetch sys.argv[1] and copy the string except for first two characters
    if option == 'version':
        print 'Version 1.00'
    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 and exits
    --help    : Prints this help and exits'''
    else:
        print 'Unknown option.'
    sys.exit()
else:
    for filename in sys.argv[1:]:
        readfile(filename)
```

Output

```
$ python cmdline.py --nonsense
Unknown option.

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

$ python cmdline.py --version
Version 1.00

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

How It Works

This command works basically like the cat command familiar to Linux/BSD/Unix users. You just specify the names of some text files and this command will print them to the output for you.

When Python is not run in interactive mode (i.e. you are not using the interpreter prompt), there is always at least one item in the sys.argv list which is the name of the current Python program being run. This is accessed by sys.argv[0] since Python starts counting from 0. Similarly, if there is one command line argument to our program, then sys.argv will contain two items and sys.argv[1] refers to the second item.

To make the program more user-friendly, we have supplied certain options that the user can use to learn more about the program. We use the first argument to check if any options have been specified to our program. If the --version option is used, the version number of the command is printed. Similarly, when the --help option is used, some explanation about the program is printed. We make use of the sys.exit() function to exit the running program. You can optionally return a status code using this function. As always, see help(sys.exit) for details.

When only filenames are specified, the program then prints out the files, one by one, to the standard output. As an aside, the name cat is short for concatenate which is basically what this program does i.e. it can print out a file or attach (concatenate) two or more files together in the output.

More sys

The sys.version string gives you information about the version of Python that you have installed. The sys.version_info tuple gives an easier way of enabling Python-version specific components of your program.

```
$ python
>>> import sys
>>> sys.version
'2.2.3 (#1, Oct 15 2003, 23:33:35) \n[GCC 3.3.1 20030930 (Red Hat Linux 3.3.1-6)]'
>>> sys.version_info
(2, 2, 3, 'final', 0)
>>>
```

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

For all this and more, please see the Python Standard Documentation. Yes, see it right now.

The os module

This module represents operating system specific functionality. This module is especially important if you want to make your programs platform-independent i.e. it should run on Linux as well as Windows without any problems and without requiring changes. An example is using the os.sep string instead of '\\' path separator in Windows, the '/' path separator in Linux or ':' path separator in Mac.

Some of the more useful parts of the os module are listed below. Most of them are self-explanatory.

- The os.name string specifies which platform you are using. If you are using Windows, it will say 'nt'. If you are using Linux or BSD, it will say 'posix'. If you are using the Jython interpreter instead of the CPython interpreter, it will say 'java'.

- The os.getcwd() function returns the current working directory i.e. the name of the directory from which the current Python script is running.

- The os.getenv() and os.putenv() functions are used to get and set environment variables.

- The os.listdir() function returns the name of all files and directories in the specified directory.

- The os.remove() function is used to delete a file.

- The os.system() function is used to run a shell command.

- The os.linesep string gives the line terminator used in the current platform. For example, Windows uses '\r\n', Linux uses '\n' and Mac uses '\r'.

- The os.path.split() function returns the directory name and file name of the path.

```
>>> os.path.split('/home/swaroop/poem.txt')
('/home/swaroop', 'poem.txt')
```

  The os.path.isfile() and the os.path.isdir() functions check if the given path refers to a file or directory respectively. Similarly, the os.path.exists() function is used to check if a given path actually exists.

You can explore the Python Standard Documentation for more details on these functions and variables. You can use the help() as well.

Summary

We have seen some of the functionality of the sys and os modules in the Python Standard Library. You should explore the Python Standard Documentation to find out more about these and other modules as well.

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

Chapter 15. More Python

Table of Contents

Till now, have covered the majority of the aspects of Python that you will use. In this chapter, we will cover some more aspects that will make our knowledge of Python complete.

Special Methods

There are certain special methods for classes that have some special semantics.

Examples are the __init__ and __del__ methods which we have already used. Generally, special methods are used to mimic certain behavior. For example, if you want to use the x[key] indexing operation for your class just like you use for lists and tuples, then you just implement the __getitem__() method and your job is done. If you think about it, this is what Python does for the list class itself!

Some useful special methods are listed in the following table. If you want to know about all the special methods, then a huge list is available in the Python Reference Manual.

Table 15.1. Some Special Methods

Name	Explanation
__init__(self, [..])	This method is called just before the newly created object is returned for usage.
__del__(self)	Called just before the object is destroyed.
__str__(self)	Called when we use the print statement with the object or when str() is used.

Name	Explanation
__lt__(self, other)	Called when the less than operator (<) is used. Similarly, there are special methods for all the operators (+, >, etc.).
__getitem__(self, key)	Called when the x[key] indexing operation is used.
__len__(self)	Called when the built-in function len() 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 it's own indentation level. Well, this is true for the most part but it is not entirely true. If your block of statements contains only one single statement, then you can specify it on the same line of, say, a conditional statement or looping statement.

The following example should make this clear.

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

As we can see, the single statement is used in-place and not as 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. One reason is that it will be much easier to add an extra statement if you are using proper indentation.

Note
Also notice that the Python interpreter when used in the interactive mode, helps you when you enter the statements. In the above case, after you entered the keyword if, it changes the prompt to ... to indicate that the statement is not yet complete. When we do complete the statement in this manner, we press enter to confirm that the statement is complete. Then Python runs the statement and returns to the old prompt that we have become familiar with.

List Comprehensions

List comprehensions are used to derive a new list from an existing list. For example, you have a list of numbers and you want to get a corresponding list with all the numbers multiplied by 2 but only when the number itself is more than 2, then list comprehensions are ideal for such situations.

Using List Comprehensions

Example 15.1. Using List Comprehensions

```
#!/usr/bin/python
# Filename: list_comprehensions.py

listone = [2, 3, 4]
listtwo = [2*i for i in listone if i > 2]
print listtwo
```

Output

```
$ python list_comprehensions.py
[6, 8]
```

How It Works

Here, we derive a new list by specifying the manipulation to be done (2*i) when some condition is satisfied (if i > 2). Note that the original list remains unmodified. Many a time, we use loops to process each element of a list - the same can be achieved using list comprehensions in a more precise, compact and explicit manner.

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 to the parameter name respectively. This is useful for receiving a variable number of arguments in a function.

```
>>> def sum(number, *args):
...     '''Return the sum the number of args.'''
...     total = 0
...     for i in range(0, number):
...             total += args[i]
...     return total
...
>>> sum(3, 10, 20, 30)
60
>>> sum(2, -5, -10)
-15
>>>
```

Due to the * prefix on the args variable, all the extra arguments passed to the function are stored in args as a tuple. If a ** prefix had been used instead, the extra parameters would have been stored in a dictionary.

Lambda Forms

A lambda statement is used to create new function objects and then return them.

Using Lambda Forms

Example 15.2. Using Lambda Forms

```
#!/usr/bin/python
# Filename: lambda_form.py

def make_repeater(n):
    return lambda s: s * n

twice = make_repeater(2)
twicetheword = twice('word')
print twicetheword
```

Output

```
$ python lambda_form.py
wordword
```

How It Works

Here, we use a function make_repeater to create new function objects at runtime and then return it. A lambda statement is used to create the function object which is invoked just like any other function. Essentially, the lambda statement is a function generator. Note that the lambda form's content must be a single expression only - it cannot even be a statement like the print statement.

The exec statement

The exec statement is used to execute Python statements which are stored in a string or file. For example, we can generate a string containing Python code at runtime and then execute these statements using the exec statement. A simple example is shown below.

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

The eval statement

The eval statement is used to evaluate valid Python expressions which are stored in a string. A simple example is shown below.

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

The assert statement

The assert statement is used to assert that something is true. For example, if you are very sure that you will have at least one element in a list you are using and you want to check this and make sure an error is raised if it is not true, then, the assert statement is ideal in this situation. When the assert statement fails, an AssertionError is raised.

```
>>> mylist = ['a']
>>> assert len(mylist) >= 1
>>> mylist.remove('a')
>>> mylist
[]
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
>>>
```

The repr function

The repr function is used to obtain a canonical string representation of the object. Backticks (also called conversion or reverse quotes) does the same thing. Note that you will have eval(repr(object)) == object most of the time.

```
>>> i = 5
>>> `i`
'5'
>>> repr(i)
'5'
>>>
```

Basically, the repr function or the backticks are used to obtain a printable representation of the object. You can control what your objects return for the repr function by defining the __repr__ method in your class.

Summary

We have covered some more features of Python in this chapter and yet you can be sure we haven't covered all the features of Python. However, at this stage, we have covered most of what you are ever going to use in practice. This is sufficient for you to get started with whatever programs you are going to create.

Next, we will discuss how to explore Python further.

Chapter 16. What Next?

Table of Contents

If you have read this book thoroughly till now and practised writing a lot of programs, then you must have become very comfortable and familiar with Python. You have probably created some Python programs to try out some stuff and to exercise your Python skills as well. You have already seen how to create a backup script as well. The question now is "What Next?"

I would suggest that you tackle this problem : create your own command-line address-book program using which you can add, modify, delete or search for your contacts such as friends, family and colleagues and their information such as email address and/or phone number. Details must be stored for later retrieval.

This is fairly easy if you think about it in terms of all the various stuff that we have come across till now. If you still want directions on how to proceed, read the following hint. (Hint: Create a class to represent the person's information. Use a dictionary to store person objects with their name as the key. Use the cPickle module to store the objects persistently on your hard disk. Use the dictionary built-in methods to add, delete and modify the persons.)

Once you are able to do this, you can claim to be an accomplished programmer. Now, immediately send me a mail thanking me for this great book ;-). This step is optional but highly recommended!

Here are some ways to continue your journey with Python:

Graphical Software

GUI Libraries using Python - you need these to create your own graphical programs using Python. You can create your own Winamp or IrfanView (Windows users should be familiar with these) or your own XMMS or KOrganizer (Linux/BSD users should be familiar with these) using these GUI (graphical user interface) libraries with their Python bindings. Bindings are what allow you to write programs in Python using libraries which themselves are written in C or C++ such as all the GUI libraries we mention below.

There are lots of choices for GUI using Python:

- PyQt. This is the Python binding for the Qt toolkit which is the foundation upon which KDE is built. Qt is extremely easy to use and very powerful because of the excellent Qt Designer and the amazing Qt documentation. You can use it for free on Linux but you will have to pay for it if you want to use it on Windows. PyQt is free if you want to create free (GPL'ed) software and paid if you want to create proprietary software. A good resource on PyQt is GUI Programming with Python .

- PyGTK. This is the Python bindings for the GTK+ toolkit which is the foundation upon which GNOME is built. GTK+ has many quirks in usage but you can become really productive with it once you get used to it. The Glade graphical interface designer is indispensable. The documentation is yet to improve. GTK+ works well on Linux but it's port to Windows is incomplete. You can create both free software as well as proprietary software using GTK+.

- wxPython. This is the Python bindings for the wxWidgets toolkit. wxPython has a learning curve associated with it but it is very portable and runs on Linux, Windows, Mac as well as some embedded platforms and all this without any changes to your code. There are many excellent IDEs for wxPython which include GUI designers as well, such as the Boa Constructor and the spe (Stan's Python Editor) as well as the free wxGlade GUI builder. You can create free as well as proprietary software using wxPython.

- PythonCard. Many readers have suggested that PythonCard is a very good alternative for basic GUI programs. It is actually a layer over wxPython.

- TkInter. TkInter is one of the oldest GUI toolkits. If you have used IDLE, you have seen a TkInter program at work. The documentation for TkInter at PythonWare.org is comprehensive. TkInter is portable and works on both Linux as well as Windows.

Summary of GUI Tools

Unfortunately, there is no one standard GUI tool for Python. I suggest that you choose one of the above tools depending on your situation. The first factor is whether you are willing to pay to use any of the GUI tools. The second factor is whether you want the program to run on Linux or Windows or both. The third factor is whether you are a KDE or GNOME user on Linux.

Explore More

- The Python Standard Library is an extensive library. Most of the time, this library will have what you are looking for. This is referred to as the "batteries included" philosophy of Python. I highly recommend that you go through the [Python Standard Documentation] that comes with your Python installation before you proceed to start writing large Python programs.

- Python.org - The official site of Python with comprehensive information regarding Python. You will also find the latest versions of Python here. There are links to various mailing lists where active discussions about Python take place.

- comp.lang.python. This is the Usenet newsgroup where discussion about this language takes place. You can post your doubts and queries to this newsgroup as well. You can access this online using Google Groups.

- Python Cookbook is an extremely valuable collection of recipes or tips on how to solve certain kinds of problems using Python. This is a must-read for every Python user.

- Charming Python is an excellent series of Python-related articles by David Mertz.

- Dive Into Python is a very good book for experienced Python programmers. If you have thoroughly read the current book you are reading ("A Byte of Python"), then I would highly recommend that you read "Dive Into Python" next. It covers a range of topics including XML Processing, Unit Testing, and Functional Programming.

- IBM DeveloperWorks - you will find interesting articles such as Grid Computing using Python .

- O'Reilly Network - here you will find the latest Python news.

- Python at O'Reilly - this is an exhaustive resource containing many Python-related articles.

- Google - the ultimate search engine for the ultimate dynamic programming language (Python)! Also take a look at the Google Jobs website which lists Python as one of the pre-requisites for qualification for a software engineering job at Google.

- Jython - this is an implementation of the Python interpreter in the Java language. This translates to using the expressiveness of the Python language with the extensive Java libraries. For example, you can create Swing GUI applications using Python code.

- IronPython is an implementation of the Python interpreter in .Net/Mono. This translates to using the Python language along with the powerful .Net libraries. IronPython is from the same person behind Jython - Jim Hugunin.

- Lython is a LISP frontend to the Python language. It is similar to Common LISP and compiles directly to Python bytecode which means that it will interoperate with your usual Python code as well.

Summary

We have now come to the end of this book but, as they say, this is the beginning of the end !. You are now an avid Python programmer and user and you are no doubt ready to solve all your problems using Python. You can start automating your computer to do all kind of previously unimaginable things or write your own games or just do some plain boring backups. All this and more is possible using Python. So, get started!

Appendix A. Free/Libre and Open Source Software (FLOSS)

FLOSS is based on the concept of a community which itself is based on the concept of sharing, and particularly the sharing of knowledge. FLOSS is free for usage, modification and redistribution.

You are already familiar with FLOSS since you have been using Python all along! If you want to know more about such FLOSS, you can explore the following list. I have listed some major FLOSS as well as those FLOSS which work on both Linux and Windows so that you can try out these software without the need to switch to Linux immediately although you eventually will ;-).

- Linux. This is a free and open-source operating system that the whole world is embracing! It was started by Linus Torvalds as a student. Now, it is giving competition to Windows. It is a full-featured kernel and the new 2.6 kernel is a major breakthrough with respect to speed, stability and scalability. [Linux Kernel]

- Knoppix. This is a distribution of Linux which runs off the CD! There is no need to install it. You can reboot your computer, pop this CD in the drive and then start using a full-featured Linux distribution! You can use all the various FLOSS that come with a standard Linux distribution - you can run your Python programs, compile C programs or even burn CDs using Knoppix (of course, you will have to have two separate drives for this). Then, reboot your computer, remove the CD and then you are back to Windows as if nothing happened at all. [Knoppix]

- Fedora. This is a project sponsored by Red Hat which is the standard Linux distribution. It contains the the Linux kernel, the X Window System, the KDE and GNOME desktop environments and the plethora of FLOSS they provide and all this in an easy-to-use and easy-to-install manner. If you are a complete beginner, then I would recommend that you try Mandrake Linux. The newly released Mandrake 10 Linux is just too awesome for words. [Fedora Linux, Mandrake Linux]

- OpenOffice.org. This is an excellent complete office suite based on the Sun Microsystems' StarOffice software. You can use OpenOffice to open and edit MS Word and MS Powerpoint files as well as it's own open and excellent XML-based formats. This is the one-stop shop for all your office needs. It runs on both Linux and Windows. The upcoming OpenOffice 2.0 has some radical improvements to it. [OpenOffice]

- Mozilla Firefox. This is the next generation web browser which is predicted to beat Internet Explorer (in terms of market share only ;-) in a few years. It is blazingly fast and has gained critical acclaim for its sensible and impressive features. It works on Linux, Windows, Mac OS and many other platforms. [Mozilla Firefox]

- Mono. This is an open source implementation of the Microsoft .NET platform which allows .NET applications to be created and run on Linux, Windows, FreeBSD, Mac OS, as well as other platforms. Mono implements the ECMA standards - Microsoft, Intel and HP have submitted the CLR (Common Language Runtime) and the C# language to ECMA (European Computer Manufacturers' Association) which has accepted them as open standards. This is a step in the direction of ISO standards on the lines of ISO C/C++.

  Currently, there is a complete C# compiler mcs (which itself has been written in C#!), a feature-complete ASP.NET implementation, many ADO.NET providers for database servers and many many more features being added and improved everyday. [Mono, ECMA, Microsoft .NET]

- Apache. This is a popular open source web server. In fact, it is the most popular web server on the planet. It runs nearly 60% of the websites out there! Yes, that's right Apache handles more websites than all the competition (including Microsoft IIS) combined. It runs on Linux as well as Windows. [Apache]

- MySQL. This is an extremely popular free and open source database server. It runs on both Linux and Windows. [MySQL]

- MPlayer. This is the video player for Linux. It can play anything from DivX to MP3 to Ogg to VCDs and DVDs. Who says open source ain't fun? [MPlayer]

- Movix. This is a Linux distribution which (like Knoppix) runs off the CD but uses MPlayer to play movies from your CD. You can even create eMovix CDs which are bootable CDs - just pop in the CD into the drive, reboot the system and the movie starts playing by itself! You don't even need a hard disk to play eMovix CDs. [Movix]

This list just gives a brief idea - there are many more free and open software and technologies out there such as the Perl language, PHP language, PostNuke content management system, PHProjekt groupware, PostgreSQL database server (a real

workhorse and very reliable), TORCS racing game, KDevelop IDE, the famous Anjuta IDE (by the famous Indian Mr. Naba Kumar), XVid codec, Xine - the movie player, Apache Software Foundation projects such as the Xerces parser and the Cocoon, ... this list just could go on forever.

For an incredibly huge list of FLOSS, see the following websites.

- SourceForge
- FreshMeat
- KDE
- GNOME

To get the latest buzz in the free and open source world, see the following websites.

- OSNews
- LinuxToday
- NewsForge
- g2swaroop.net

If you want to get the latest Linux distributions and open source software on CDs, then see the following websites.

- LinCDs
- LinuxCentral

So, go ahead and explore the vast, free and open world of FLOSS!

Appendix B. About

Table of Contents

Colophon

All the software that I have used in the creation of this book have been free and open source software. In the first draft, I had used Red Hat 9.0 Linux as my setup and for this fifth draft, I have used Fedora Core 1 Linux as the foundation of my Linux box setup.

Initially, I used KWord to create this book (as explained in the History Lesson in the preface). Later, I switched to DocBook XML using Kate but I found it too tedious. So, I switched to OpenOffice which was just excellent with the level of control on the formatting as well as the PDF generation, but it produced very sloppy HTML from the document. Finally, I discovered XEmacs and I rewrote this book from scratch in DocBook XML (again). This book would not have been possible if it wasn't for XEmacs (and XMMS, of course).

The standard XSL stylesheets that come with the Fedora Core 1 Linux distribution were used. The standard default fonts were used as well. However, I have written a CSS document to give color and style to the pages. I have also written a crude lexical analyzer (in Python of course!) which automatically provides syntax highlighting to all the programs listed in this book (this applies to the HTML form of the book only).

About the Author

Swaroop C H is a final year graduate student of Computer Science at PESIT, Bangalore, India. His interests on the technological side include FLOSS such as Linux, Mono, Qt and MySQL, great languages like Python, C and C#, writing stuff like this book and any software he can create in his spare time, as well as writing his blog and maintaining his website. His ultimate goal is to create a software that will output any software that he wants or thinks. He also thinks that self-written introductions to authors are cool.

Do not forget to visit my website www.g2swaroop.net - there are lots of cool stuff here such as the latest versions of this book, many software I have written, my techstuff pages as well as my blog and my photos.

About LinCDs.com

LinCDs.com is a service started by my friend Yashwanth and myself. We are both final year students of B.E. Computer Science at PESIT, Bangalore, India.

The idea behind LinCDs.com was to make Linux accessible to everybody. We were finding it difficult to get the latest versions of distributions such as Red Hat and Mandrake and we found so many people in the same situation. We used to eventually get those CDs but after a lot of searching. So we decided to do something about it. We collected some distributions and started to provide them to others at a reasonable cost. We also started acquiring new releases quickly as well.

Now, we have more than 35 different distributions and application CDs from Fedora, Mandrake, Debian, Connectiva, ELX and other Linux distributions to FreeBSD, NetBSD and OpenBSD to GNUWin, Freeduc and other application CDs.
LinCDs.com caters to both Indian and international customers. Please see our website for more details.

Feedback

Any suggestions, praise, comments or criticisms are most welcome. You can contact me at python@2swaroop.net .