

Getting Started With



Python

Ideal for data scientists, data analysts, and developers



by:

Jose Luis

Roberto Asuncion

&

Gerald Britton

Big Data University BOOK SERIES

GETTING STARTED WITH

Python

A book for the community by the community



JOSE LUIS ROBERTO ASUNCION,
GERALD BRITTON

Second Edition (May 2015)



This work is licensed under a [Creative Commons Attribution-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nd/4.0/).

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “[Copyright and trademark information](https://www.ibm.com/legal/copytrade.shtml)” at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Table of Contents

[Chapter 1 – Quick Start](#)

[1.1 Hello World!](#)

[1.2 Installing and setting up Python](#)

[1.3 Data Scientist Workbench](#)

[1.11 Summary](#)

[Chapter 2: Fundamentals](#)

[2.2 Documentation](#)

[2.2.1 Comments](#)

[2.2.2 Docstrings](#)

[2.3 Basic Syntax](#)

[2.3.1 Assignment Statements](#)

[2.3.3 Other Arithmetic Operators:](#)

[2.3.4 Conditional Execution](#)

[2.3.5 Compound arithmetic expressions](#)

[2.3.6 Compound Logical Expressions](#)

[2.3.7 Expression chaining](#)

[2.3.8 Everything has a Boolean Value:](#)

[2.3.9 Special Operations:](#)

[2.3.10 Repetition](#)

[2.4 Exercises for Chapter 2](#)

[Chapter 3 – Functions](#)

[3.1 The Big Picture](#)

[3.2 The Built-in Functions](#)

[3.2.1 Examples of Built-in Functions](#)

[3.2.1 File Operations](#)

[3.2.2 Lists and Tuples](#)

[3.2.2.2 Built-in methods for use with lists and tuples](#)

[3.2.2.3 Comprehensions](#)

[3.2.3 Formatting Output](#)

[3.3 The Standard Library](#)

[3.3.1 The math module](#)

[3.3.2 The OS Module](#)

[3.3.3 The Platform Module](#)

[3.3.4 The sys module](#)

[3.4 User Defined Functions](#)

[3.4.1 Basics](#)

[3.4.2 Generators and Generator Expressions](#)

[3.4.3 Closures](#)

[3.4.4 Decorators](#)

[3.4.5 Co-routines and Consumers](#)

[3.5 Lambda Expressions](#)

[3.6 Exercises](#)

[Chapter 4 – Sequences, Mappings and Collections](#)

[4.1 Sequences](#)

[4.2 More fun with strings](#)

- [4.2.1 Multi-line strings](#)
- [4.2.2 Other String Methods](#)
- [4.2.3 More on lists](#)
- [4.2.4 Updating and deleting list items](#)
- [4.2.5 Operators and methods for lists and tuples](#)

[4.3 Sets](#)

- [4.3.1 Introduction to Python Sets](#)
- [4.3.2 Set operations](#)
- [4.3.3 Set Testing](#)
- [4.3.4 Sets as relations](#)
- [4.3.5 Set Methods](#)
- [4.3.6 Immutable Sets](#)

[4.4 Dictionaries](#)

- [4.4.1 Building dictionaries](#)
- [4.4.2 Building dictionaries with list comprehensions](#)
- [4.4.2 Modifying Dictionary Entries](#)
- [4.4.3 Dictionaries as Keyed Relations](#)
- [4.4.4 Iterating Over a Dictionary](#)
- [4.4.5 Dictionary Methods](#)
- [4.4.6 Using Dictionaries to Categorize and Count](#)
- [4.4.7 Using defaultdict](#)

[4.5 Named Tuples](#)

[4.6 Double-ended Queues](#)

[4.7 Exercises](#)

[Chapter 5 – Modules](#)

[5.1 The Big Picture](#)

[5.2 Introduction](#)

[5.3 Definition](#)

[5.4 Importing Modules](#)

[5.5 Executing Modules as scripts](#)

[5.6 The dir\(\) function](#)

[5.7 Packages](#)

[5.7.1 Importing modules from packages](#)

[5.7.2 Modules referencing from other modules](#)

[5.8 How does the interpreter go about looking for your files?](#)

[Chapter 6 – Object Oriented Programming](#)

[6.1 The Big Picture](#)

[6.2 Object Oriented Programming versus Procedural Programming](#)

[6.3 Classes](#)

[6.3 Creating an Object](#)

[6.4 Attributes](#)

[6.5 Behaviours](#)

[6.6 Inheritance](#)

[6.6.1 Extending Created Classes](#)

[6.7 Exercises](#)

[Chapter 7 – Database Connectivity](#)

[7.1 The Big Picture](#)

[7.2 Prerequisites](#)

[7.3 Setting up](#)

[7.4 Using IBM DB2 with Python](#)
[7.5 Connection Objects](#)
[7.6 Cursor Objects](#)
[7.7 Performing SQL queries](#)
 [7.7.1 SELECT](#)
 [7.7.2 INSERT](#)
[7.8 Exercises](#)
[Appendix A – Solutions to the review questions](#)
[Chapter 2](#)
[Chapter 3](#)
[Chapter 4](#)
[Chapter 6](#)
[Chapter 7](#)
[Web sites](#)
[Books](#)

Preface

Keeping your skills current in today's world is becoming increasingly challenging. There are too many new technologies being developed, and little time to learn them all. The Big Data University Book Series has been developed to minimize the time and effort required to learn many of these new technologies. The books that are part of this series have corresponding free online courses in BigDataUniversity.com, so you can learn using just the book, just the online course, or both!

This book is intended for anyone who works with or intends to develop Python applications such as application developers, consultants, software architects, data scientists, instructors and students. It is a good reference as well for dev ops, system administrators and product managers.

This book was created by the community; a community consisting of university professors, students, and professionals (including IBM employees). The online version of this book is released to the community at no-charge. Numerous members of the community from around the world have participated in developing this book, which will also be translated to several languages by the community. If you would like to provide feedback, contribute new material, improve existing material, or help with translating this book to another language, please send an email of your planned contribution to admin@bigdatauniversity.com with the subject "Getting Started with Python book feedback."

If you are interested in Python because you want to become a data scientist, or work with data analytics, make sure to review bigdatauniversity.com, which has a large number of free courses on these subjects.

About the authors

Jose Luis Roberto Asuncion

Jose is a code quality freak. He plays barbie with objects by dressing them up with design patterns. He is a PHP developer by day. One day he would like to focus working more with Java and Python rather than with PHP. Drop him a line at twitter.com/jeunito

Gerald Britton

Gerald Britton is currently a database developer with TD Bank and an open source contributor to various Python-based projects, including Gramps and OpenLP. Gerald has been active in database development for four decades and has worked both in North America and overseas helping companies and charities with their database needs across a variety of platforms, including Linux, Windows and OS/390. He is also a strong advocate for the open source software development model with a dedication to finding better, faster and cheaper ways of handling difficult problems.

Contributors

The following people edited, reviewed, provided content, and contributed significantly to this book.

Contributor	Company / University	Position / Occupation	Contribution
Polong Lin	University of Toronto	MSc. Psychology	Addition of content related to the Data Scientist workbench. Technical and English review and editing
Antonio Cangiano	IBM Toronto Lab	Software Developer and Technical Evangelist	Technical review
Raul F. Chong	IBM Toronto Lab	Senior Program Manager	Review and editing

1

Chapter 1 – Quick Start

Welcome to Python! This book aims to get you started on learning Python as quickly as possible. No previous programming experience is assumed. We will show you a basic Python program and then we will get you started on creating your own!

1.1 Hello World!

Let's create our first Python program. Listing 1.1 has the Python code.

```
"""This is my first program in Python YAY!"""  
print "Hello, World!"
```

Listing 1.1 – The “Hello World!” program

The output of the program, as you may have guessed is “Hello, World!” The first line in the listing is simply a comment.

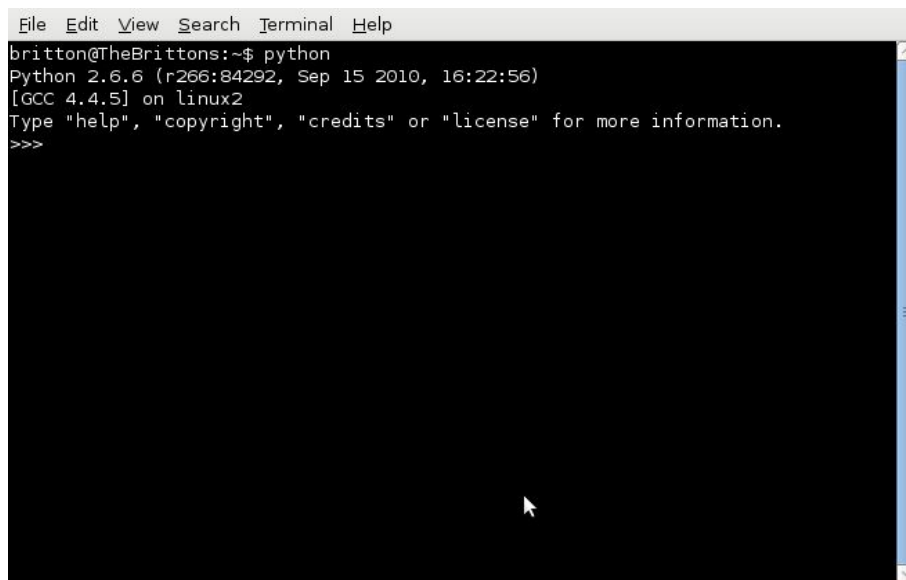
To run the program on interactive mode, start a command prompt (on Windows, if you have Python installed already) or terminal (on Linux/Mac OS) and type:

```
>>> print "Hello, World!"
```

Python is very simple. It has been designed so that the code is easy to read and is as understandable as everyday English.

1.2 Installing and setting up Python

The first step to work with Python is to download and install it for your platform. If you're using Linux or Mac, Python is already installed on your system. For example, on Linux, you can verify Python is installed by opening a terminal window, typing “python” and pressing enter. If correctly installed, output similar to the one shown in *Figure 1.1* should appear.

A terminal window with a menu bar (File, Edit, View, Search, Terminal, Help) and a title bar. The terminal text shows the command 'python' being executed, resulting in the Python 2.6.6 version and build information, the GCC version, and the operating system (linux2). It also displays the help text for the interpreter. The prompt 'britton@TheBrittons:~\$' is visible at the top.

```
File Edit View Search Terminal Help
britton@TheBrittons:~$ python
Python 2.6.6 (r266:84292, Sep 15 2010, 16:22:56)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1.1 – Verifying if Python has been installed on Linux

For Windows users, follow these steps to get Python up and running on your computer:

1. Download the Python interpreter from <http://www.python.org/downloads/> and choose the installer that fits with your operating system. Install the latest release of version 2.7. (As of this writing, version 2.7.9 is current). This is shown in Figure 1.2.

Note: For the purposes of this introduction, we are not using Python 3.0. As a language, Python 3.0 has several attractive features and enhancements. However, for the widest compatibility with the many commercial and open source libraries that are available, we will stick with Python 2.7.

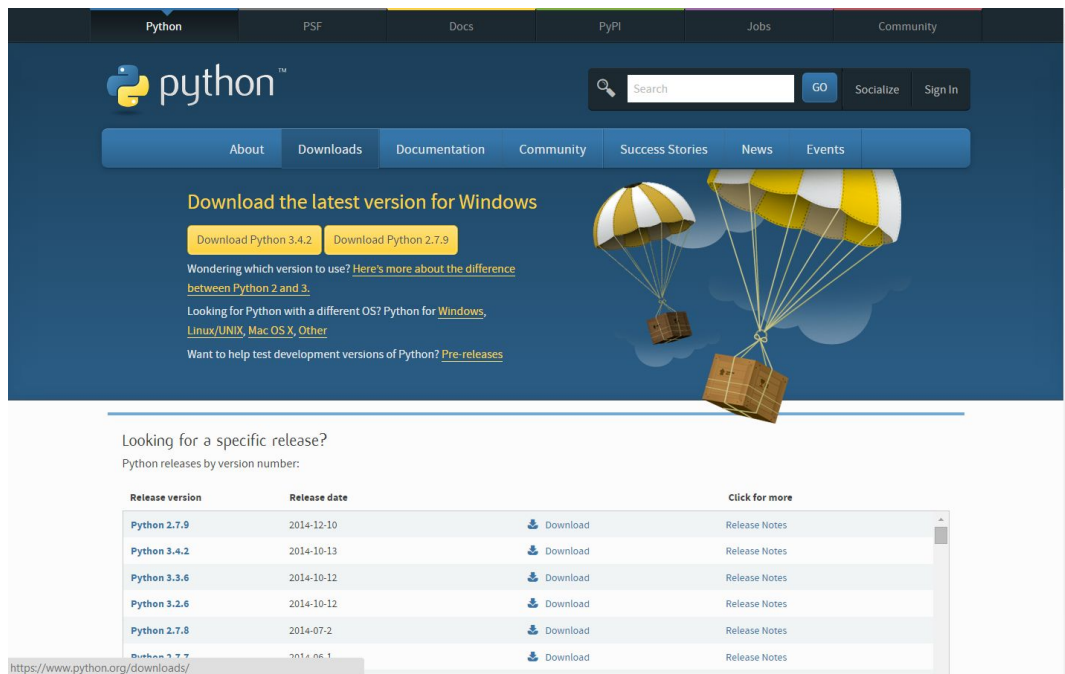


Figure 1.2 – Downloading Python

2. After selecting the Python version to download, a .msi file is downloaded. Launch it directly after the download completes. If prompted, give the installer permission to make changes to your computer. Figures 1.3 through 1.6 show the installation panels you will see. They are self-explanatory.



Figure 1.3 – Installing Python for all users?

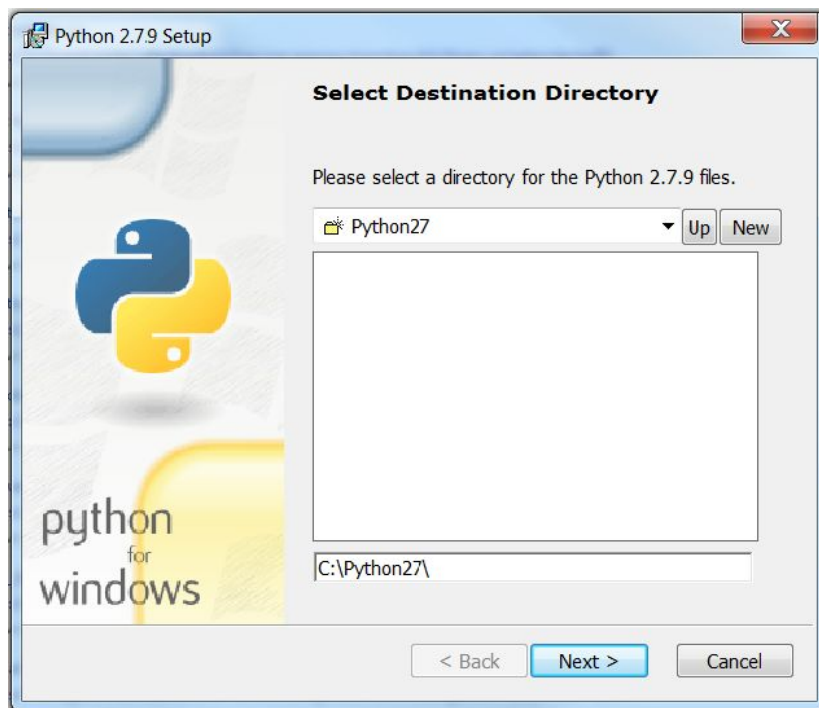


Figure 1.4 – Select Destination Directory



Figure 1.5 – Customize Python

Be sure to enable the option “Add python.exe to Path”

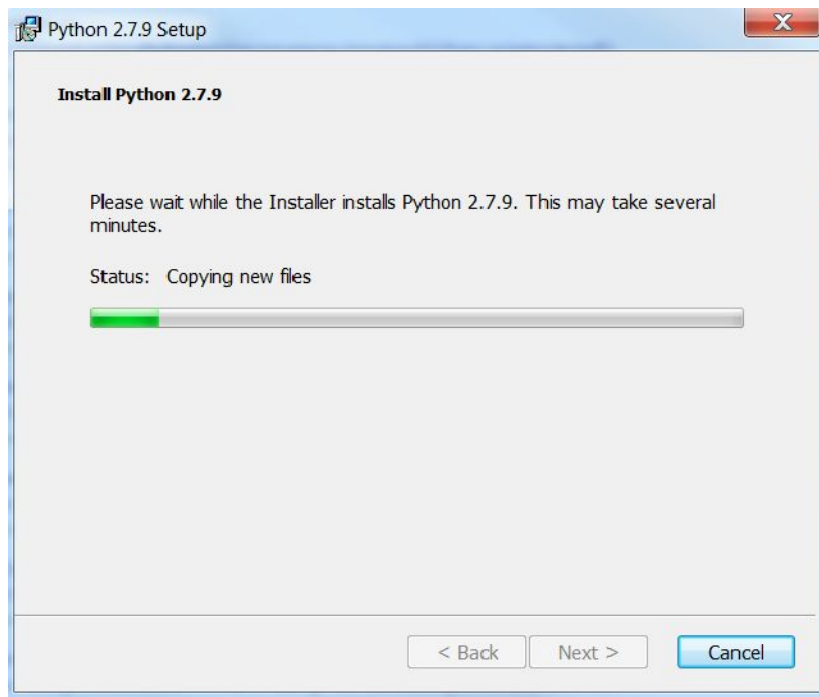


Figure 1.6 – Installing Python

3. Test the installation by opening a command prompt and typing: `python`. (You may need to log out and log back in to Windows again for the change to the Path environment variable to take effect.) When you reach the Python prompt, type

```
print "Hello world!"
```

and hit the Enter key.

1.3 Data Scientist Workbench (datascientistworkbench.com)

Although you can certainly get started with Python on its own, you may find yourself wanting extra features to help you save your scripts, organize your projects, and display your output all in the same environment. Fortunately, there's the Data Scientist Workbench (DSWB) at <https://datascientistworkbench.com>. Powered by iPython, a command shell for interactive computing, DSWB lets you write, run and save Python code, and even display graphics within each document or "notebook". DSWB is hosted on IBM's cloud, allowing you to access your Python notebooks from any computer. The interface is customizable, and lets you organize your projects and data files with ease. Best of all, like Python, it's free!

To follow along the Python code in this textbook, you can use Python on its own, but we recommend taking a look into DSWB if you plan on working with Python in the future. They have some excellent tutorials to help get you started using DSWB.

1.11 Summary

It is very easy to get started with Python. On many systems (including Linux and OSX) the language is pre-installed. For Windows users, the installation is quick and painless!

2

Chapter 2: Fundamentals

Python was designed with several specific goals in mind:

- Simple implementation (Guido van Rossum wrote the original version entirely on his own)
- Very high-level language
- Cross-platform portability (write once, run anywhere)
- Automatic memory management
- Everything is an object
- Small number of powerful data types
- Readability and expressive power
- Easy to learn and remember
- Predictability
- Safety (bugs in a Python program do not crash the interpreter)
- Extensibility with C-language modules

Those goals have been achieved and maintained throughout its more than twenty years history. Today there are many contributors who continue to refine and extend the language.

In this chapter, we will look at some of the fundamental things needed to become an effective Python programmer. Let's dive in and put Python to work. That means you need to know a little syntax. We'll start off with a few basics, but before all else, let's be sure we do *one* thing right: documentation!

2.2 Documentation

Virtually all professional programmers would agree that a program without proper documentation is incomplete, regardless of the language. Some languages are quite verbose (e.g. COBOL, which claims to be "self-documenting" to a degree but still can be used to develop inscrutable programs.) Others are extremely terse (perhaps APL is the best example). No matter what language is used to write it, a program without documentation will very likely defy understanding and maintenance. Python is no exception; fortunately it provides ample facilities to embed documentation within each program. As we go along, we'll show you how to do that and "practice what we preach" as well.

2.2.1 Comments

The most basic type of documentation is an in-line comment. In Python, a comment begins with a # symbol (variously called a number sign or a hash mark or symbol, depending on where you live) and finishes with the end of the line. This is an example of a comment:

```
# This is a comment
```

A comment can start anywhere on a line, and take place even after other Python commands. Many comment lines in a row can be used to introduce the method or purpose of a larger block of code that follows. Get in the habit of using both sorts of comments -- a lot! You'll thank yourself later, when you go back to your program and can't remember why you did what you did or the way you did it. If someone else "inherits" your program, that person will be especially thankful.

2.2.2 Docstrings

A docstring (short for documentation string) is used to document the purpose, input arguments and return arguments of a function, class or method. It begins and ends with three double-quotation marks. Lines between the beginning and end do not need the quotation marks. This is an example of a docstring:

```
"""
This is an example of a docstring
"""
```

We'll start to use these in earnest when we begin to work on user-defined functions in Chapter 3.

2.3 Basic Syntax

In this section we will look at the building-blocks of a Python program -- the elements you will need to begin to write real programs. We'll also be needing a few types of data. For now, we'll stick with just three:

- Integers
 - o An integer is just, well, an integer! 42, for example.
- Floating point numbers
 - o A floating point number usually has a decimal point, like 3.14159
- Strings
 - o A string is a sequence of characters enclosed in quotations -- either single, double, triple-single or triple-double. e.g. 'Hello', "world", "I can use triple opening and closing quotations.", ""Python is fun!"". If you are coming from a language like C you may be used to using double-quotations for strings and single-quotations for characters. That is actually a good convention and a one that we will try to stick with.

2.3.1 Assignment Statements

Most programming languages have some sort of an assignment statement. That is a command -- specific to the language -- that says that some variable name can be used to represent some constant or expression (which may involve other variable names). For example, in mathematics, we can assign the product of two or more numbers or variables to a variable name:

$$x = 2 \times y \quad (1)$$

If we assign the value 2 to the variable y :

$$y = 2 \quad (2)$$

and then compute equation (1) with this, we should expect the variable x to receive the value of 4. Let's do this in Python! Start the Python interpreter in an interactive mode following the instructions for your operating system. If you are running Linux, you might start it like this:

```
$ python
Python 2.7.9 (default, Dec 10 2014, 12:24:55)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The ">>>" is the interpreter prompt and this is where you can begin entering Python statements. We can enter equations (2) and (1) directly, except that the multiplication symbol in Python (as in many other languages) is the asterisk (*):

```
>>> y = 2                # set y to the integer 2
>>> x = 2 * y            # compute 2 times the variable y
>>>
```

The equals sign (=) is recognized by Python as the assignment operator. That is, when the equals sign appears between two operands, the meaning is to assign the value of the second operand to the first one. So, in the first statement, the value 2 is assigned to the variable y ; in the second statement the value of the expression $2 * y$ is assigned to the variable x . (This is probably a good time to point out that, in Python, x and y in this example do **not** receive the values 2 and 4, respectively, but rather references or pointers to storage locations where 2 and 4 are stored. This is a subtle but important distinction between Python and some of the other languages with which you may be familiar.)

Before we go further, let's get one thing out of the way: identifiers. An identifier is a name you give to something, like a variable. So in the above example, "x" is an identifier. In Python, you can make identifiers from any sequence of letters, numbers and underscores ("_") except that the first character *must* be a letter or underscore. Try to make your identifiers say something about what they are identifying. For example, "day" is better than "i" if you're talking about a day of the week or month.

Notice that, as each statement is entered, Python responds with a new prompt, which means that it is ready for input again. Notice also that we used spaces to separate the variable names, numbers, the multiplication symbol and the "equals" sign. Spacing like this is not required in Python but is heartily encouraged. There is even a special document, (called Python Enhancement Proposal number 8, or simply PEP8) that goes into great detail about what good Python style is and is not. In this book, we will attempt to

follow the guidelines of PEP8 and other style guides. We want you to develop good habits from the very start!

Always surround binary operators with a single space on either side.

2.3.2 Checking Results

So far, so good. At this point we have to check whether our math worked and if `x` really got the value 4. To verify this, simply type the variable name `x` and hit enter:

```
>>> x
```

```
4
```

```
>>>
```

Python displays the current value of the variable `x` on the terminal. Notice that, in interactive mode, it is not necessary to use the `print` command for this, though that works just as well:

```
>>> print x
```

```
4
```

```
>>>
```

2.3.3 Other Arithmetic Operators:

There are other ways to perform various arithmetic operations in Python. For starters, here is a table showing the Python operators that correspond to basic arithmetic operations:

Mathematical Operator	Python Equivalent	Example	Result	Operation
$x + y$	<code>x + y</code>	<code>2 + 1</code>	3	Addition
$x - y$	<code>x - y</code>	<code>2 - 1</code>	1	Subtraction
$x \times y$	<code>x * y</code>	<code>2 * 1</code>	2	Multiplication
$x \div y$	<code>x / y</code> <code>x // y</code>	<code>2 / 1</code> <code>3.0 / 2</code> <code>3 // 2</code>	2 1.5 1	Integer division Real division Integer division
$x \text{ modulo } y$	<code>x % y</code>	<code>5 % 2</code>	1	Remainder of division
x^y	<code>x ** y</code>	<code>2 ** 3</code>	8	Exponentiation

The different types of division in this table demands an explanation. Python recognizes two types of division: integer and real. With integer division, both the dividend and the divisor are treated as integers and the result is also an integer. That means that, when the quotient is not an integer, it is rounded *down* to the closest integer. With real division, the dividend and divisor are treated as real numbers and the quotient is also a real number without rounding. The `//` (double-slash) operator explicitly specifies that you want integer

division. We would encourage you to use the double-slash operator when you want integer division and not rely on Python's ability to choose the type of division based on the types of the dividend and divisor. This practice adheres to another Python principle, "Explicit is better than implicit" -- one of nineteen such principles found in PEP20, entitled "The Zen of Python".

Explicit is better than implicit!

For example:

```
>>> 2/3
0
>>> 2.0/3
0.66666666666666663
>>>
```

2.3.4 Conditional Execution

Let's say that you wanted to divide one number by another, but you had no idea in advance what the divisor would be. Chances are, at some time the divisor would be zero! In mathematics, the operation of dividing by zero is undefined, so you would want to avoid doing that and maybe tell the user about the problem.

That's easy to do in Python using the if statement. Let's assume that you are given two variables, x and y and you need to divide x by y and avoid dividing by zero. One solution would be:

```
>>> x = 2
>>> y = 3
>>> if y != 0:
...     print x / y          # only attempt division if the divisor is non-zero
... else:
...     print "I can't divide by zero!"
...
0
>>>
```

This code snippet demonstrates the use of the if statement. You begin with the word "if", and follow it with some condition ("y != 0" in this example), and further follow that with a colon. The operator "!=" is the Python equivalent of the \neq operator from mathematics. The use of the colon in this context is a consistent pattern in Python and a one that will be encountered many times. In this context, it implies that, "Only do the next section of code if the condition above is true." In this case, the next section of code is the line "print x / y". Notice that this line is indented four spaces from the if statement that precedes it. Indentation is *required* here and indicates that all the indented lines are part of a suite of Python code that is dependent upon the if statement. The use of four spaces for indentation is one of the style guidelines in PEP8, which we are following. Notice also that the indentation gives a visual clue that the print statement is subordinate to the if statement.

Use 4 spaces per indentation level.

The else statement tells Python what to do in the case where y is, in fact, zero. (Notice that the else statement is followed by a colon and that the next line is indented four spaces.) Let's try the same example with y set to zero:

```
>>> x = 2
>>> y = 0
>>> if y != 0:
...     print x / y          # only attempt division if the divisor is non-zero
... else:
...     print "I can't divide by zero!"
...
I can't divide by zero!
>>>
Python did exactly what it was told it to do. It avoided the division by zero
and informed us of it as well.
```

There are many situations where a simple "either-or" such as the above example is not up to the job.

Suppose you wanted to print a different phrase depending on the day of the week. Let's assume you are given a variable, day_number, that is an integer from 1 to 7 where 1 means Monday, 2 means Tuesday, ... and 7 means Sunday. To do this, we will use the elif statement:

```
>>> day_number = 4          # Let's try Thursday!
>>> if day_number == 1:
...     print "Monday's child is fair of face"
... elif day_number == 2:
...     print "Tuesday's child is full of grace"
... elif day_number == 3:
...     print "Wednesday's child is full of woe"
... elif day_number == 4:
...     print "Thursday's child has far to go"
... elif day_number == 5:
...     print "Friday's child is loving and giving"
... elif day_number == 6:
...     print "Saturday's child works hard for a living"
... elif day_number == 7:
...     print "But the child born on the Sabbath Day"
...     print "Is fair and wise and good and gay"
... else:
...     print "There are only seven days in a week!"
...
Thursday's child has far to go
>>>
```

The elif statement gives you a convenient way to line up various conditions so that they appear visually as a unit. Some languages provide a similar function through a "switch" or "case" statement; "elif" is used in Python to do a similar thing.

2.3.4 Other Conditional Operators

Python includes a full suite of conditional operators. The following table shows Python operators and the corresponding mathematical operations:

Mathematics Operator	Python Equivalent	Example	Result	Operation
$x = y$	<code>x == y</code>	<code>1 == 2</code>	False	Equals
$x < y$	<code>x < y</code>	<code>1 < 2</code>	True	Strictly less than
$x \leq y$	<code>x <= y</code>	<code>1 <= 2</code>	True	Less than or equal to
$x > y$	<code>x > y</code>	<code>1 > 2</code>	False	Strictly greater than
$x \geq y$	<code>x >= y</code>	<code>1 >= 2</code>	False	Greater than or equal to
$x \neq y$	<code>x != y</code> <code>x <> y</code>	<code>1 != 2</code> <code>3 <> 4</code>	True True	Not equal to

Note that there are two operators that can be used to test the "not equal to" condition. This is a convenience provided by Python for programmers familiar with other languages (including SQL). You can choose whichever one you prefer, but *be consistent!*

2.3.5 Compound arithmetic expressions

So far, we've only looked at very simple expressions. Things are not always so simple! What if you were working on quadratic equations? For example, the general equation for a parabola with a vertical axis is:

$$y = ax^2 + bx + c \quad (3)$$

Python allows you to build a compound expression from simpler ones, so the translation of this equation into Python is not difficult:

```
y = a * x ** 2 + b * x + c    # Compute the y-axis coordinate of a quadratic equation
```

(Go ahead and try this in an interactive Python session using your own values for a, b, c and x.)

By looking at the Python statement more closely, you may wonder how it "knows" that you want it to do the exponentiation first, then the multiplication and follow that with addition. After all the results would be quite different (and wrong!) if the computation proceeded strictly from left-to-right or from right-to-left (Prove this to yourself with values you choose for a, b, c and x!) Python solves this by obeying a system of *precedence* for mathematical operations. For the arithmetic operators we have seen so far, operations are done according to this table, with the top row having the highest precedence and the bottom row having the lowest.

Operator	Description
(expression)	parentheses
**	exponentiation
*, /, //, %	multiplication, division, modulus
+, -	addition, subtraction
+x, -x	positive, negative

Notice that this table introduces operators that we have not seen before: positive (+x) and negative (-x). These are called *unary* operators since they only affect one variable. Note also that white space is neither required nor recommended between a unary operator and its target so, for example:

```
+a * -b
```

is correct but:

```
+ a * - b
```

is not recommended (and harder to read in the opinion of the authors!)

What if you actually need to override Python's built-in precedence of operations? You can do that by using parentheses. Expressions inside parentheses are always calculated first, from the inmost parenthesized expression to the outermost. We could force the (incorrect!) left-to-right evaluation of equation (3) using parentheses, like this:

```
# Use parentheses to force strict left-to-right evaluation
y = (((a * x) ** 2 + b) * x) + c
```

This is saying: Multiply a by x, then square the result; then add b and multiply the new result by x; finally, add c. As an exercise, how would you use parentheses to force right-to-left evaluation of equation (3)?

2.3.6 Compound Logical Expressions

We've seen the way to build compound arithmetical expressions. Now what if we need to test for some combination of conditions? For example, let's say that you want to play tennis with a friend, but you will only do that if the temperature is greater than 80 degrees (Fahrenheit) and there is more than 70% sunshine. If we have two Python variables, "temp" and "sunshine", we could do it like this:

```
if temp > 80 and sunshine > 70:
    ...
```

Here we have two conditions ("temp > 80" and "sunshine > 70") joined by the keyword "and". The suite of statements (represented by the ellipsis, "...") following the "if" statement will not be executed unless both conditions are true. If you are a little more flexible and will accept either condition, you could write:

```
if temp > 80 or sunshine > 70:
    ...
```

We can also express the same idea negatively: "I'll play tennis as long as the temperature is not less than 80 and there is not less than 70% sunshine."

```
if not (temp < 80) and not (sunshine < 70):
    ...
```

Here, we combine the use of parentheses the "and" keyword and introduce the "not" keyword as well. Note that this expression *violates* two rules from PEP20, "Simple is better than complex" and "Beautiful is better than ugly".

<i>Simple is better than complex.</i>
--

<i>Beautiful is better than ugly.</i>
--

Following these two rules, we can see that the first compound expression is the better one. Now that we have three new operators, we need to put them in our precedence table so that we know what to expect. The new precedence table looks like this:

Operator	Description
(expression)	parentheses
**	exponentiation
*, /, //, %	multiplication, division, modulus
+, -	addition, subtraction
+x, -x	positive, negative
<, <=, ==, !=, <>, >, >=	comparisons
not	Boolean NOT
and	Boolean AND
or	Boolean OR
if...else	Conditional expression

Note: We will continue the discussion of conditional expression further in this chapter.

2.3.7 Expression chaining

Python offers shorter expressions for some comparisons. For example, suppose you want to execute some instructions if the variable named "fred" is less than 0 and is also equal to another variable, "nancy". You could write it this way:

```
if fred < 0 and fred == nancy:
```

but Python has a shorthand expression that will do the same thing:

```
if 0 < fred == nancy:
```


If that looks a little odd to you, you are not alone! In fact, Python allows unlimited chaining of comparisons so that long expressions such as:

```
if a < b == c <= d >= e:
```

are valid! The advantages are three-fold:

1. You only have to write each variable once (provided that you can chain them together as above and still make logical sense)
2. Python only has to evaluate each variable once, if at all.
3. If the expression evaluates false at any point, moving from left-to-right, the remainder of the expression is not evaluated.

The last advantage is the result of Python's *short-circuiting* strategy. Basically, Python will only evaluate what it needs to in order to proceed, saving extra work and avoiding possible errors. For example, if the variables "buffalo" and "bill" are not defined, the following statement will work just fine:

```
>>> if 1 > 2 < buffalo < bill:
...     print "shouldn't get here"
...
>>>
```

Since the first comparison "1 > 2" evaluates to False, the variables "buffalo" and "bill" are never evaluated, which is a good thing since they are undefined! Try this on your own and just for fun, change the > to < and observe the new results. If they are different, can you explain why?

2.3.8 Everything has a Boolean Value:

In Python, everything -- every variable, number, function, collection, sequence, ... -- (some of which you haven't even seen as yet), has a boolean value. That means that you can put anything in a conditional to see if it is True or False. As mentioned before, everything in Python is also a pointer. This includes the integer "1" - which is a pointer to a storage location that holds the integer.. When you write:

```
if var:
```

you are essentially asking Python to look at whatever "var" points to (an operation called de-referencing) and determine the Boolean value of that. If "var" happens to point to a number, Python considers it False if the number is equal to zero and true otherwise. "var" could refer to either of the Boolean constants True or False (which are also keywords that you can use), in which case the value is returned directly.

Often, "var" will refer to some other type of Python object, or None. We capitalize "None" here because None is a special Python constant that is unique for the lifetime of the program in which it is referred to. If var points to None, it is considered False in a Boolean sense. Though we haven't discussed them yet, it is a good time to note that empty collections and sequences are considered False and a non-empty other collection or sequence is considered True.

2.3.9 Special Operations:

There are a few special operations that should be mentioned at this point. First up, we will discuss the augmented assignment statements. An augmented assignment is a short-hand way of modifying a value referred to by some variable. For example, a common programming operation is to increment (or decrement) a counter by one.

Typically this is done like this:

```
x = x + 1
```

but I can use an augmented assignment statement to shorten this to:

```
x += 1
```

The obvious advantage to using an augmented assignment statement is that there is a lot less to type. This is not such a big deal for this example. However, if the variable name was "monty_pythons_flying_circus", chances are that you would want to type that as little as possible. There is a more subtle, behind the scenes benefit. With the standard assignment statement, "x" is actually evaluated twice; using the augmented assignment means it is evaluated only once. On the whole, it is less work for you, less work for Python and less work for the computer you run it on. Use it!

Another special operation allows us to shortcut some kinds of tests. Consider an if statement that is used to effect a variable assignment:

```
if a:                # set c based on whether a is True or False
    c = a
else:
    c = b
```

This is a common pattern found in many programs. In Python, this can be shortened to:

```
c = a and b
```

Python interprets this one-line statement so that the result (the assignment of the value of either "a" or "b" to "c") is the same as the full if-else statement above. That is, if "a" is True (according to the discussion above under "Everything has a Boolean value", "c" is now identified with the same value as "a", otherwise "c" is identified with the same value as "b".

Note: In Python, we use the word "binding" to describe the operation of identifying a variable with some value. Thus, in the example above, "c" is bound to whatever "a" or "b" are bound to, respectively. The binding operation is accomplished by using pointers to particular storage locations -- that is, memory addresses.

Similar to the above example:

```
c = a and b
```

means that "if a evaluates to False, bind c to whatever a is bound to (because of short-circuiting), otherwise bind c to whatever b is bound to." Here is an example of this short-hand in action:

```
>>> 0 and 2 < 3
0
>>> (0 and 2) < 3
True
```

In the first statement, following the rules of precedence, " $2 < 3$ " is evaluated first; the result is True. That means that the next expression to be evaluated is "0 and True". Following the rules just given, 0 evaluates to False, so 0 is returned. In the second statement, we force the Boolean operation to be evaluated first using parentheses. "0 and 2" evaluate to 2 following the rules just given. " $0 < 3$ " evaluates to True, which is what Python returns! As an exercise, change the "and" to "or" in the examples above and convince yourself that Python really behaves as described in this section.

The last special operation that we will look at in this section is the conditional expression. A conditional expression is one that uses if...else to return a single value. For example:

```
a = 1 if b else 2                                # Set a to either 1 or two depending on b
```

This is the same as the longer:

```
if b:                                             # Set a to either 1 or two depending on b
    a = 1
else:
    a = 2
```

Conditional expressions are a useful shorthand that can be used anywhere as long as you are careful about the rules of precedence. Try variations of these two expressions in an interactive Python session and convince yourself that the interpreter is behaving correctly in each case. (Remember that the conditional expression fits immediately below the "or" operator in the operator precedence table.)

```
>>> 1 + 2 if False else 4
4
>>> 1 + (2 if False else 4)
5
```

2.3.10 Repetition

There are not too many interesting programs that do useful things and still run in a straight line from start to finish. Most of the programs involve some kind of a repetition. This can vary from doing math on a given value to waiting for . There are two Python statements that can be helpful for these common sorts of operations. These are: "for" and "while".

Here is an example of a simple "for" loop in action:

```
>>> for i in 1, 2, 3:
...     print i
...
1
2
3
>>>
```

All this loop does is to print the numbers 1, 2 and 3 in succession. Nevertheless, it demonstrates the key elements of a “for” loop. Notice that the general syntax is:

```
for <identifier> in <things>:
    <suite>
```

The keyword “for” is a variable name that is to be used in the loop. Next comes the keyword “in” which gives you a hint that something else should follow. Finally we have the things that the identifier will be used to reference. This can be a simple list such as in the example or an identifier that refers to something that can be iterated over. Lastly comes the all-important colon. After the for statement comes the body of the loop -- the suite of instructions that are executed as part of the “for” loop. In this example, there is only one instruction:

```
    print i
```

which is indented 4 spaces from the for statement that precedes it following the guidelines of PEP8.

“For” loops can be nested and frequently must be in order to perform useful work. For example:

```
>>> for day in "Mon", "Tue", "Wed":
...     for letter in day:
...         print letter
...
M
o
n
T
u
e
s
d
>>>
```

The outer loop iterates over the shortened names of the first three days of the week. The inner loop iterates over the letters in each of those names. (You will learn more about strings and how to take them apart in the chapter on Sequences and Collections). Notice the indentation levels of the outer and inner loops.

Now let's take a look at a simple "while" loop:

```
>>> n = 5
>>> f = 1
>>> while n > 0:
...     f = f * n
...     n -= 1
...
>>> f
120
>>>
```

What is this loop doing? It is computing $n!$ (n factorial) which is the product of all the integers up to and including a given integer. The general syntax is:

```
while <condition>:
    <suite>
```

The loop will only begin if the condition is true and will only continue as long as the condition remains true. A "while" loop may also be nested if desired.

There are two situations that arise within loops that skip to the end of the loop or exit it before it is finished. The first is accomplished with a continue statement and the second through a break statement. For example, suppose you have a for loop that takes in some integers, but you only want to process these numbers if they are even:

```
>>> for i in 1,2,3,4,5:
...     if i % 2:                # If i mod 2 is not zero, go get the next number
...         continue
...     print i
...
2
4
>>>
```

What if you only want to print some letters of the string "Wednesday"?

```
>>> for letter in "Wednesday":
...     if letter == "s":
...         break
...     print letter
...
W
e
d
n
e
>>>
```

Notice how the break statement causes the loop to stop before it processes all of the letters in "Wednesday". As another example, suppose you want to take in some number and print out the numbers between it and 100:

```
>>> number = 105
>>> while True:
...     print number
...     if number >= 100:                # If we've reached 100, stop now
...         break
...     number -= 1
...
105
104
103
102
101
100
>>>
```

Notice that the loop starts with "while True:" This means that there must be some other way of stopping the loop, or it will run forever.

2.4 Exercises for Chapter 2

1. Each of the following Python statements has one or more errors. Correct the errors and test your corrections in an interactive Python session:

a. `y = 2 x 3` # Multiply 2 by 3 and assign the result to the variable y

b. `if 4 = 2/2:` # Check if 4 is equal to 2/2
 `print "4 = 2/2!"`

c. `for x in 1 2 3:` # print the first three positive integers on three lines
 `print x`

d. `x = 4 if b` # assign 4 to x if b is True, otherwise leave it alone

e. `while True:`
 `x = 1`
 `if x: continue`

2. Fibonacci numbers are defined with this function:

$$F_n = F_{n-1} + F_{n-2}$$

with seed values:

$$F_0 = 0 \quad F_1 = 1$$

Write a Python program fragment that computes the Fibonacci number for a given number n, and then prints the result.

3. It is possible to approximate the sine of any random value x (in radians) using this Taylor series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \text{ for all } x$$

Write a Python program fragment that performs this calculation to six terms (of which the first three are given) and compare your results to the published sine tables.

4. Write a Python program fragment that, when given a month of the year as an integer between 1 and 12, prints out the month name, the number of days in the month (assume February always has 28 days) and also the name of the birthstone assigned to the month.

3

Chapter 3 – Functions

3.1 The Big Picture

For most programming languages today, functions are where the action happens. For one thing, functions provide the interface between the program and the real world. Without functions, every programmer would have to write low-level code to read from a file, print data on a printer, manage the graphics on a monitor or get what a user types at a keyboard. Functions also give us the ability to solve a problem once and then use that solution everywhere. Imagine that you always had to write the code necessary to calculate a square root or sort names or add up a bunch of numbers every time you wanted to do that. You would probably quickly get tired of such repetitive work and the chance of getting it wrong would increase each time you had to write the same (or only slightly-different) code again.

In this chapter we will begin by exploring some of the functions in the standard library. This library provides a deep store of highly-optimized functions for an array of typical tasks. We will also look at one of Python's most pervasive data types -- lists -- since many functions (and their close cousins, methods) are designed to work with them. We will also learn how to write our own functions and how to use recursion and closures to make our lives simpler.

3.2 The Built-in Functions

Whenever you run Python, you have an immediate access to a set of very-useful functions -- so useful that they are built-in and you can use them without further ado. To try them out, you need to call them. Calling a function in Python is very simple:

```
>>> function(arguments)
```

3.2.1 Examples of Built-in Functions

Let's try out a few built-in functions to see what they do:

```
>>> abs(-1)           # Absolute value
1
>>> bin(3)            # Convert to binary
'0b11'
>>> chr(97)           # Convert to character (inverse of ord())
'a'
```



```

>>> hex(26)           # Convert to hexadecimal
'0x1a'
>>> int('26')         # Convert a string to an integer
26
>>> len("Hello World") # Return the length of the argument
11
>>> max(1, 2)          # Maximum of arguments
2
>>> min('a', 'b', 'c') # Minimum of arguments, which don't have to be numbers!
'a'
>>> oct(35)           # Convert to octal
'043'
>>> ord('a')          # Convert to an integer (inverse of chr())
97
>>> str(44)           # Convert to a string
'44'
>>> bool(1)           # Return True or False according to Boolean evaluation
True

```

Currently, there are nearly 150 built-in functions in Python. Refer to the official documentation for a complete, up-to-date list of built-in functions. There is one built-in function that is arguably the most useful of all: `help(x)`. Try these statements in an interactive Python session:

```

help(help)

help(int)

help(max)

help(bool)

```

As you can see, it produces one or more screens of documentation about the item you are requesting help for. The argument to `help()` can be any Python object. Since, in Python *everything* is an object, you can request help about anything at all, even operators. For example, try:

```
help('+')
```

or just:

```
help           # Start interactive help
```

3.2.1 File Operations

One important group of built-in functions warrants special discussion is the group used to access files. Let's start off by using the built-in function `open()` to create a file and get it ready for writing:

```
>>> myfile = open("my.txt", 'w')
That's all it takes! Now the file "my.txt" is open for writing. Let's fill it up with some data:
```

```
>>> i = 0
>>> while i < 5:
...     myfile.writelines("Line number " + str(i))
...     i += 1
...
>>> myfile.close()
```

What is going on here? Well, when the `open()` function was called, it returned a reference to a file object. Like all objects in Python (even basic objects like integers and strings), it has "methods" associated with it that you use to manipulate the object. Under the covers, methods are actually functions that are designed to work with a particular type of object. In this example, we used two methods that are part of the file object (there are several others). In the while loop, we used the write method to write five lines to the file we just opened. (We used the `str()` built-in function to convert the loop counter to a string.) When the loop finished (that is, when the loop counter became equal to 5), we used the `close()` method to close the file.

Did it work? Well, let's read it and see:

```
>>> myfile = open('my.txt', 'r')
>>> while True:
...     line = myfile.readline()
...     if line == '':
...         break
...     print line
...
Line number 0
Line number 1
Line number 2
Line number 3
Line number 4
>>> myfile.close()
>>>
```

This time, when we opened the file, we specified that it was to be opened for reading. (This is actually the default action so we could have omitted the argument 'r'.) Next, we entered a while loop with the condition set to `True`. That means that we need to have something else to stop the loop or it will be infinite. Within the loop, we use the `readline()` method of the file object to read the next line and assign it to the variable "line." Next, we check to see if that line is just an empty string, "". The `readline()` definition states that an empty string is returned when EOF (end of file) is reached. If we have indeed reached EOF, we break out of the while loop. Otherwise, we print the data we received from the `readline()` method.

There is another, somewhat more convenient way to read the lines from our file. We can do it like this:

```
>>> myfile = open('my.txt', 'r')
>>> for line in myfile:
...     print line
...
Line number 0
Line number 1
Line number 2
Line number 3
Line number 4
>>> myfile.close()
>>>
```

This approach uses the fact that the file object includes an "iterator" that automatically returns all the lines in the file and stops at the end of file, without our having to explicitly look for EOF. Many objects in Python have iterators. We saw one in Chapter 2: strings. When we have a for-loop like this:

```
>>> for letter in "word":
...     print letter
...
w
o
r
d
>>>
```

What we are really doing is using the built-in iterator for strings, which returns the string one character at a time. There is another, very important type of data in Python that uses iterators and for which several built-in functions are designed: lists.

3.2.2 Lists and Tuples

Lists (and their stuffy, read-only cousins, tuples) are pervasive in Python. You would be hard-pressed to find any serious Python program that does not use lists in some fashion. Defining a simple list or tuple is easy to do:

```
>>> mylist = ["this", "is", "my", "list"]
>>> mytuple = ("this", "is", "a", "tuple")
```

The brackets (for lists) or parentheses (for tuples) are required here and tell Python to build a list or tuple and return a reference to it. The main difference between a list and a tuple is that a tuple is *immutable* -- that is, it may not be changed once it is created. What can I do with a list? Well, for starters, I can loop over the items in a list like this:

```
>>> for item in mylist:
...     print item
...
```

```
this
is
my
list
>>>
```

Single items in the list can also be referred to by number. The first item is 0, the second 1 and so on. To specify which item we want, we put the item number in brackets (the characters [and]). We can also count from the end, using negative numbers, so -1 is the last item, -2 is the penultimate item and so on. Here is one way to print out the contents of mylist backwards:

```
>>> i = -1
>>> while i >= -4:
...     print mylist[i]
...     i -= 1
...
list
a
is
this
>>>
```

It is also possible to take pieces out of the list, using a notation called *slicing*. For example:

```
>>> print mylist[1:3]
['is', 'a']
>>>
```

`mylist[i:j]` is a slice that selects all items with index k such that $i \leq k < j$. Thus, "list" is not included in `mylist[1:3]`.

3.2.2.2 Built-in methods for use with lists and tuples

Many built-in functions are designed to operate on lists, tuples and other types that support iteration. These include:

```
>>> all([0, 1, 2])      # Returns True if all list items evaluate to True
False
>>> any([0, 1, 2])      # Returns True if any list item evaluates to True
True
>>> len([0, 1, 2])      # Returns the number of items in the list
3
>>> min([0, 1, 2])      # Returns the smallest item in the list
0
>>> max([0, 1, 2])      # Returns the largest item in the list
2
```

```
>>> sum([0, 1, 2])          # Adds up all the items in the list
3
>>> range(5)                # Returns a list of integers from 0 to 4
[0, 1, 2, 3, 4]
>>> map(str, range(5))      # Returns a list of integers converted to strings
['0', '1', '2', '3', '4']
>>> filter(bool, range(5))  # Returns a list of only those items that evaluate
to True
[1, 2, 3, 4]
>>>
```

3.2.2.3 Comprehensions

Python offers a concise way to build lists and other objects using *comprehensions*. Basically, Python is able to understand a special kind of a for-loop within a list or tuple definition (and others, including sets and dictionaries) and build a list or tuple according to the results returned by the loop. For example, if I wanted a list comprised of the squares of the first 100 integers, I could use a list comprehension to do it:

```
>>> [i**2 for i in range(100)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, ..., 9801]
>>>
```

Note: We truncated the display to make it fit comfortably in this document.

The list comprehension says, "For each integer from 0 to 99, insert an item into the list equal to the square of that integer." To see just how powerful this syntax is, consider the way to do the same thing with other tools. Perhaps it would look something like this:

```
>>> squares = []
>>> for i in range(100):
...     squares.append(i**2)
...
>>>
```

This will accomplish the same thing but it takes three lines to do it and the for-loop contains several variable look ups. It is also about 60% slower. Here are a few more complicated examples showing the power of comprehensions:

```
>>> [(i,j) for i in range(5) for j in range(2)]
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1), (4, 0), (4, 1)]
>>>
```

It is possible to nest comprehensions. Here we used this facility to construct a list consisting of tuples.

```
>>> [(i,j) for i in range(5) for j in range(2) if i + j < 3]
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0)]
>>>
```

We can add conditions on the comprehension to limit what is included. There is no limit on what the conditions can entail, provided that they are syntactically valid.

```
>>> tuple([i,j] for i in range(5) for j in range(2) if (i + j) % 3)
([0, 1], [1, 0], [1, 1], [2, 0], [3, 1], [4, 0], [4, 1])
```

Here we use a comprehension to build a tuple containing items that are lists, provided that the sum of the items in each list is not a multiple of 3. Notice that we must use the built-in function `tuple()` to do this. Otherwise Python would build a generator, which we will cover later in this chapter.

3.2.3 Formatting Output

Sooner or later, you will want to exert some control over the appearance of the output of your program. Perhaps you will need to write a tabular report and ensure that the columns line up properly. Perhaps your output will be the input to another program that requires a particular format. Perhaps you simply want to add some text around the results of your calculations. You can do a fair bit with the humble print statement:

```
>>> print "The square of", 2, "is", 2**2
The square of 2 is 4
```

What if you want to print a table of squares and ensure that numbers and their squares are all right aligned? Just using the print statement alone, you would be forced to add or remove spaces to force the alignment, since the numbers or their squares might have multiple digits. We need a better, more consistent approach, and that is where the `format()` function and the string function of the same name come to the rescue.

First let us see if we can use the `format()` function to solve the problem of alignment mentioned in the preceding paragraph. We can do this:

```
>>> for i in range(2,12,2): # range() will make a list starting at 2, ending at
...     print "The square of", format(i, ">3d") , "is", format(i**2, ">3d")
...
The square of    2 is    4
The square of    4 is   16
The square of    6 is   36
The square of    8 is   64
The square of   10 is  100
>>>
```

Notice how the numbers in the output are right aligned? This is what the `format()` function did for us. The first argument given is the value we want to format (the "what"). The second argument is the format specification (the "how"). The specification is written in a special mini-language that is derived from similar output formatting facilities found in C and other languages. A format specifier is a string (thus enclosed in quotation marks, as in the example above) that is constructed from 0 or more elements in the following order:

```
[[fill] align] [sign] [#] [0] [width] [.precision] [type]
```

where:

Option	Meaning										
fill	A fill character, which can be any character other than '}'. If a fill character is specified, then the next character must be one of the alignment options.										
align	<table><tr><th>Option</th><th>Meaning</th></tr><tr><td>'<'</td><td>Left aligned (default)</td></tr><tr><td>'>'</td><td>Right aligned (as in our example)</td></tr><tr><td>'='</td><td>Put padding after the sign, if any, but before the first digit.</td></tr><tr><td>'^'</td><td>Center aligned</td></tr></table>	Option	Meaning	'<'	Left aligned (default)	'>'	Right aligned (as in our example)	'='	Put padding after the sign, if any, but before the first digit.	'^'	Center aligned
Option	Meaning										
'<'	Left aligned (default)										
'>'	Right aligned (as in our example)										
'='	Put padding after the sign, if any, but before the first digit.										
'^'	Center aligned										
sign	<table><tr><th>Option</th><th>Meaning</th></tr><tr><td>'+'</td><td>Use a sign for both positive and negative numbers.</td></tr><tr><td>'-'</td><td>Only use a sign for negative numbers. (This is the default behaviour.)</td></tr><tr><td>space</td><td>Use a leading space on positive numbers and a minus sign on negative numbers</td></tr></table>	Option	Meaning	'+'	Use a sign for both positive and negative numbers.	'-'	Only use a sign for negative numbers. (This is the default behaviour.)	space	Use a leading space on positive numbers and a minus sign on negative numbers		
Option	Meaning										
'+'	Use a sign for both positive and negative numbers.										
'-'	Only use a sign for negative numbers. (This is the default behaviour.)										
space	Use a leading space on positive numbers and a minus sign on negative numbers										
#	Only for integers in binary, octal or hexadecimal output. Prefixes the output by '0b', '0O', or '0x' respectively										
0	Pad the output with a zero.										
width	Minimum field width. If not specified, then the field width is determined by the content										
precision	Number of digits to be displayed after the decimal point, for floating point numbers. Not valid for integers. For non-numbers, it indicates the maximum field width.										
type	<table><tr><th>Option</th><th>Meaning</th></tr><tr><td>'s'</td><td>String format. This is the default for string values and may be omitted</td></tr><tr><td>'b'</td><td>Binary. Formats a number in base 2</td></tr><tr><td>'d'</td><td>Decimal. Formats a number in base 10</td></tr><tr><td>'o'</td><td>Octal. Formats a number in base 8</td></tr></table>	Option	Meaning	's'	String format. This is the default for string values and may be omitted	'b'	Binary. Formats a number in base 2	'd'	Decimal. Formats a number in base 10	'o'	Octal. Formats a number in base 8
Option	Meaning										
's'	String format. This is the default for string values and may be omitted										
'b'	Binary. Formats a number in base 2										
'd'	Decimal. Formats a number in base 10										
'o'	Octal. Formats a number in base 8										

'x'	Hexadecimal. Formats a number in base 16, using lower-case letters for the digits above 9
'X'	Hexadecimal. Formats a number in base 16, using upper-case letters for the digits above 9
'n'	Number. This is the same as 'd' but inserts separators according to the locale.
'e'	Exponent notation. Uses 'e' to indicate the exponent.
'E'	Exponent notation. Uses 'E' to indicate the exponent.
'f' or 'F'	Fixed point
'g' or 'G'	General format. Prints in either fixed point or scientific format depending on the value and the specified field width. Lower case 'g' uses lower-case 'e' for scientific notation. Upper case 'G' uses upper case 'E' for scientific notation.
'%'	Percentage. Multiplies by 100, displays in fixed ('f') format and appends a percent sign.

Whew! There are lots of ways to format the data. You can use the `format()` method of the string data type to perform variable substitution and formatting on many items at once. Returning to our example, we could do it this way:

```
>>> for i in range(2,12,2): # range() will make a list starting at 2, ending at
...     print "The square of {0:>3d} is {1:>3d}".format(i, i**2)
...
The square of    2 is    4
The square of    4 is   16
The square of    6 is   36
The square of    8 is   64
The square of   10 is  100
>>>
```

Let us look at the differences in the print statement. First, there is just one string, which embeds some formatting instructions. This is followed by a period ('.') and the format function call, with arguments corresponding to the values we want to format and insert into the string for printing. Let's dissect the formatting instructions. A format string contains "replacement fields" surrounded by braces (the characters { and }). Anything *not* inside braces is copied as-is to the output. (If you need a brace in the output, include them in pairs: {{ and }}.)

Inside the braces, the replacement field is built up as follows:

```
field [! conversion] [: format]
```


where "field" is either a field name or number. Our example uses numbers, which simply means that the value for a field is the corresponding argument supplied to the format method, counting from zero. The field name may also, optionally, include a subscript for lists and tuples, keys for mapping objects (discussed in Chapter 4) and attributes for class instances (discussed in Chapter 6). "conversion" is for special cases and normally not required; it may either be "r" or "s". (See the official documentation for more details.) The "format" is the format specification we just discussed. Here are some alternate ways to print the lines in our example, showing some of the ways you can specify the field:

```
print "The square of {integer:>3d} is {square:>3d}".format(integer=i,
square=i**2)
```

```
print "The square of {0[0]:>3d} is {0[1]:>3d}".format([i, i**2])
```

3.3 The Standard Library

The Python standard library is organized into *modules*, with each module containing many functions for use in a specific area. There are about 250 modules in the standard library at the moment. Some you will use right away. Others you may never use, if your work never touches on the domains covered by those modules. Let's start off by looking at one of the most used modules: math!

3.3.1 The math module

If we want to use the math module, we need to tell Python to go and get it. We do that using the import statement. Open up an interactive Python session and type:

```
>>> import math          # Import the math module
```

Now that we have imported the math module, we can use the sin function:

```
>>> math.sin(1)
0.8414709848078965
```

This statement says, "Look up the function called 'sin' in the module 'math', then call that function with the value '1' and return the results." That is exactly what happened. (If you are skeptical, go ahead and look up the value of sine(1) in a table of sines.) The math module provides 18 trigonometric functions, six power and logarithm functions, 12 functions from number theory and two important constants. Try these functions on your computer:

Function name and arguments	Purpose	Example
<code>math.factorial(x)</code>	returns x factorial	<code>>>> math.factorial(10)</code> 3628800
<code>math.ceil(x)</code>	returns the smallest integer greater than or equal to x	<code>>>> math.ceil(3.5)</code> 4.0
<code>math.sqrt(x)</code>	returns the square root of x	<code>>>> math.sqrt(10)</code> 3.1622776601683795
<code>math.radians(x)</code>	converts angle x from degrees to radians	<code>>>> math.radians(60)</code> 1.0471975511965976
<code>math.degrees(x)</code>	converts angle x from radians to degrees	<code>>>> math.degrees(2)</code> 114.59155902616465
<code>math.pi</code>	the mathematical constant pi	<code>>>> math.pi</code> 3.1415926535897931 <code>>>> math.degrees(math.pi)</code> 180.0

3.3.2 The OS Module

The OS module contains a number of functions that can be used to query the operating system for information about the running process (where Python is running), to manipulate files, to issue commands and for many other, os-specific purposes. We will cover just a few examples that encourage you to consult the official documentation for a complete list of available functions and how to use them. Let us start by importing the module:

```
>>> import os
Now, let's try some simple queries:

>>> os.access('my.txt', os.F_OK)           # See if we can access a file
True
>>> os.rename('my.txt', 'your.txt')         # Rename the file
>>> os.access('my.txt', os.F_OK)           # See if we can still access the file
False
>>> os.getenv('HOME')                      # Retrieve the environment variable HOME
'/home/user1'
>>> os.getcwd()                            # Find the current working directory
'/home/user1'
>>> temp_file = os.tmpfile()                # Open a temporary file for writing
>>> os.system('uname -a')                  # Issue a system command
Linux ubuntu 2.6.31-20-generic #58-Ubuntu SMP Fri Mar 12 05:23:09 UTC 2010 i686
GNU/Linux
```

Try these functions on your own computer (some are platform-specific):

Function name and arguments	Purpose
<code>os.getlogin()</code>	Returns the name of the currently logged-in user.
<code>os.mkdir(path)</code>	Create a directory with the supplied path name.
<code>os.rmdir(path)</code>	Remove a directory with the specified path name.
<code>os.chdir(path)</code>	Change the current working directory.
<code>os.listdir(path)</code>	Return a list of the names of the files in the specified path.

3.3.3 The Platform Module

The platform module provides access to identification information for the system on which Python is running. At the time of writing, it included 17 functions that are platform-generic (available on all supported platforms) and another seven for specific platforms. For example:

```
>>> import platform
>>> platform.architecture()
('32bit', 'ELF')
>>> platform.machine()
'i686'
>>> platform.node()
'ubuntu'
>>> platform.platform()
'Linux-2.6.31-20-generic-i686-with-Ubuntu-9.10-karmic'
>>> platform.python_version()
'2.6.4'
>>> platform.system()
'Linux'
>>> platform.uname()
('Linux', 'ubuntu', '2.6.31-20-generic', '#58-Ubuntu SMP Fri Mar 12 05:23:09
UTC 2010', 'i686', '')
>>>
```

Try each of these on your own system and observe the results. If you are running any version of Windows, try the `platform.win32_ver()` function as well.

3.3.4 The sys module

This module provides access to some Python interpreter variables and to certain functions that are closely-aligned with the interpreter's operation. Here are a few examples from the available 70+ functions and variables:

```
$ python -i - -a=1 -b=2 -c="foo"
Python 2.6.4 (r264:75706, Dec 7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.argv          # List of command-line arguments
['-', '-a=1', '-b=2', '-c=foo']
>>> sys.maxint         # Largest supported integer
2147483647
>>> sys.maxsize        # Largest object size
2147483647
>>> sys.platform       # Platform we're running on
'linux2'
>>> sys.version        # Version of Python (compare to platform.python_version())
'2.6.4 (r264:75706, Dec 7 2009, 18:45:15) \n[GCC 4.4.1]'
```

Try each of these on your system and observe the results. Some other functions to try:

Function	Purpose
<code>sys.builtin_module_names</code>	Tuple of the names of the modules compiled with your version of Python
<code>sys.copyright</code>	Python copyright statement
<code>sys.getwindowsversion()</code>	Windows only: version information
<code>sys.modules</code>	Dictionary mapping module names to loaded modules
<code>sys.path</code>	List of paths to search for modules
<code>sys.winver</code>	Version number used for registry keys on Windows

3.4 User Defined Functions

3.4.1 Basics

Having just surveyed a few of the modules in the standard library and knowing that there are 240 or more modules that we haven't even considered-you might wonder if you would even need to define your own functions. As it turns out, there are a few serious applications that you will build that will *not* require you to write your own functions -- if only to make your programs more manageable and readable. Fortunately, writing functions is not difficult. The basic template for a function is as follows:

```
def function_name(arg1,arg2,...,keyword_arg1=default, keyword_arg2=default):  
    """  
    documentation string describing the purpose of the function,  
    what its input arguments are and what it returns  
    """  
    # body of function  
    return returned_data
```

Let us look at each of these pieces in turn. Every function definition begins with the word "def", that is followed by the name of the function. Quoting from PEP8, "Function names should be lowercase, with words separated by underscores as necessary to improve readability."

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

After the function name, with no intervening white space, comes a list of the arguments that your function will use, enclosed in parentheses. First come the positional argument names, then the keyword argument names with default values. Note that you cannot mingle the positional and keyword arguments.

Immediately following the "def" statement should come a documentation string explaining what the function is for. It should include the input arguments, if any and describe what data is returned, if any. It is indented four spaces.

Following the docstring comes the body of the function, indented four spaces from the docstring. If data is to be returned to the caller of the function, there should be a "return" statement that does just that.

Now that we understand the basic format, let's write a simple function that computes a Fibonacci number:

```
>>> def fibonacci(n):  
...     """  
...     Computes the n'th Fibonacci number  
...  
...     Input argument: n - an integer  
...     Returns: the n'th Fibonacci number  
...     """
```

```

...
...     if n == 0:
...         return 0
...     elif n == 1:
...         return 1
...     else:
...         return fibonacci(n-1) + fibonacci(n-2)
...

```

Copy this function definition (stripping out the '>>>' and '...' from the beginning of the lines) and try this function in an interactive Python session. It should return the result for any Fibonacci number up to about 1000, though it will get slower as the numbers get bigger. To understand why, we have to first look at the last line of the function. The "return" statement calls the fibonacci() function twice. This is called *recursion* and is a quite common approach for solving certain repetitive problems. However, it is not a good approach for this problem, since the overhead of using recursion makes the running time stretch out as the input numbers increase -- with no end in sight! Actually, there *will* be a stopping-point. Python has a built-in limit for how *deep* recursive calls may go. You can access that depth using the function sys.getrecursionlimit().

```

>>> import sys
>>> sys.getrecursionlimit()
1000
>>>

```

In this case, the limit is 1000, which means that it should be impossible to ask for fibonacci(1002). Practically, you would not want to, since it may take hours to compute this way! See Exercise 3.5.1 for a constant-time approach to this problem. Meanwhile, let's see if we can do it without recursion in linear time:

```

>>> def fibonacci(n):
...     """
...     Computes the n'th Fibonacci number in linear time.
...
...     Input argument: n - an integer
...     Returns: the n'th Fibonacci number
...     """
...
...     if n == 0:
...         return 0
...     fc = fn = 1
...     for _ in xrange(2, n):
...         fn, fc = fn + fc, fn
...     return fn

```

There are some things in this example that we haven't discussed before. First of all, it is permitted in Python to chain assignments, which is why the line:

```
fc = fn = 1
```

causes both `fc` and `fn` to be initialized to the value 1. Second, the for-loop uses the `xrange()` function. This function works similarly to the `range()` function discussed earlier, but serves up the numbers one at a time, rather than in a list, which saves memory and processing time. (Computer scientists call that “lazy” evaluation.) Also notice that the loop uses `'_'` as a variable name. This is a common practice when we don't need the variable for anything else. In this case, it is only used to run the mechanics of the loop. The third new idea is the assignment statement in the body of the loop:

```
fn, fc = fn + fc, fn
```

This assignment statement assigns two values to two variables. Also, because of the way Python executes statements like this, you can use the same variables on the left and on the right sides of the assignment. You can do so without worrying about the execution order. It works as if there were invisible temporary variables to hold the intermediate values (which is not far from the truth). With this new, improved function, you should be able to calculate any Fibonacci number up to the limits of your machine architecture. (In fact, you might run out of memory if you try some really large values.)

There is one other thing we should do with our function. We need to check our input argument to be sure that it is valid. It needs to be a non-negative integer. To check this, we can use the built-in function `isinstance()` like this:

```
isinstance(n, int)                # Return True if n is an integer
```

Using this, before we begin any calculations, we will add the following test to our function:

```
if not isinstance(n, int) or n < 0:
    raise ValueError, "Argument must be an integer >= 0"
```

This statement also introduces the “raise” command, which causes an exception to be raised (“ValueError” in this case) to alert the caller that the function was called with an invalid argument. Python comes with a number of built-in exceptions and you can also define your own if the need arises. As an exercise, insert the “if” statement above into the linear version of the function and test it with various invalid values.

Sooner or later, it is likely that you will need to write a function that needs to process a variable number of arguments. In fact, the string method `format()` is an example of such a function. That means that that you cannot list all of the arguments when you define the function. So, how does the `format()` method do it and how should you? The answer lies in the use of the “*identifier” and “**identifier” forms. An example of such as use would be:

```
def var_func(*args, **kwargs):
    <suite>
```

This function takes in an unknown number of positional arguments followed by an unknown number of keyword arguments. Let's see this in action:

```
>>> def func(*args, **kwargs):
...     print "Args:", args
...     print "Keyword args:", kwargs
```

```

...
>>>
>>> func(1, 2, 3, a=4, b=5, c=6)
Args: (1, 2, 3)
Keyword args: {'a': 4, 'c': 6, 'b': 5}
>>> func()
Args: ()
Keyword args: {}

```

As you can see, the positional arguments are inserted into a tuple. This can be accessed through the same name as in the function definition. Similarly, the keyword arguments are inserted into a dictionary (covered in Chapter 4) which you can access using the dictionary methods covered in Chapter 4.

3.4.2 Generators and Generator Expressions

In Section 3.4.1 we mentioned that a function must include a return statement if it is to return something to the caller. That's not quite true. There is another statement that will do that as well: `yield`. However, the presence of this statement turns the function into something special: a *generator*. A generator is a *producer* of information.

Let's see how this might work with our Fibonacci function:

```

>>> def fibonacci_generator(n=0):
...     """
...     Yields Fibonacci numbers from 0 to n
...
...     Input argument: n - an integer, the highest Fibonacci number to be
generated
...     Yields: Fibonacci numbers from 0 to n, inclusive
...     """
...
...     if not isinstance(n, int) or n < 0:
...         raise ValueError, "Argument must be an integer >= 0"
...
...     yield 0
...     if n == 0:
...         return
...     yield 1
...     fc = fn = 1
...     for _ in xrange(2, n):
...         fn, fc = fn + fc, fn
...         yield fn

```

Now, let's use this generator:

```

>>> fib_nums = fibonacci_generator(10)
>>> for f in fib_nums:
...     print f

```



```
...
0
1
2
3
5
8
13
21
34
55
>>>
```

What is going on here? First of all, let us see how we changed our `fibonacci()` function into our new `fibonacci_generator()` function. For starters, we added a default value for `n`, so that we can call our generator without any arguments and get the first Fibonacci number. Next, we added a `yield` statement to send the first Fibonacci number, 0, back to the caller. Then, assuming that the caller wants more results, we added a second `yield` statement to send the second Fibonacci number, 1. Finally, in the body of the loop, every time we calculate a new number, we use the `yield` statement to send the next Fibonacci number back to the caller. At compile time, when Python sees a `yield` statement in a function, it marks that function as a generator. At execution time, when a `yield` statement is encountered, the result of the `yield` statement is sent back to the caller and execution is suspended. The next time the caller calls, the function continues until it hits another `yield` statement, and so on, until no more `yield` statements are encountered and the function exits.

When we use our generator, the first statement is:

```
>>> fib_nums = fibonacci_generator(10)
```

merely establishes a reference to our generator, but does not start executing it. Then, the mechanics of the `for`-loop repeatedly calls a special method of the loop expression, `next()` (that is, in our example, `fib_nums.next()`) until the generator exits.

There is also another special generator method, `close()` that we can use to force a generator to stop. Here is an example of it in action:

```
>>> fib_nums = fibonacci_generator(10)
>>> for f in fib_nums:
...     print f
...     if f > 5:
...         fib_nums.close()
...
0
1
2
3
5
8
>>>
```

Python also supports a sort-of in-line generator, called a *generator expression*. It is useful for simple generators that can be written as a single expression. For example, suppose you wanted to produce squares of integers. A generator expression like this would do the trick:

```
>>> import sys
>>> sys.maxint
9223372036854775807
>>> squares = (i**2 for i in xrange(sys.maxint))          # Generator expression
yielding squares
>>> for s in squares:
...     print s
...     if s > 100:
...         squares.close()
...
0
1
4
9
16
25
```

Notice that the generator expression looks very much like a list comprehension, except that it is enclosed in parentheses instead of in brackets. That is how Python recognizes that this is a generator expression and why we cannot use parentheses in this fashion as a tuple comprehension. In this example, we made the upper limit the same as the maximum integer value for the system running the generator, yet we only generate one square at a time. In fact, if you change the parentheses to brackets, you *will* have a list comprehension, but one so big that it may not fit in your computer's memory. Try that at your own risk!

Note: Instead of `xrange(sys.maxint)` we could have used a function from the `itertools` module called `count()`. The `count()` function starts (by default) at zero, and counts forever. Our simplified generator would then look like this:

```
>>> from itertools import count
>>> squares = (i**2 for i in count())
>>>
```

This example also shows the way to import a single function from a module.

3.4.3 Closures

Sometimes you find that you will use the same function over and over again, using some but not all of the same arguments each time. Consider the `pow()` built-in function. If you enter `help(pow)` in an interactive Python session, you will see something like this:

```
pow(...)
    pow(x, y[, z]) -> number
```

With two arguments, equivalent to `x**y`. With three arguments, equivalent to `(x**y) % z`, but may be more efficient (e.g. for longs).

Now, suppose you wanted to use the three-argument form, but for your application, the last argument is always 3. It would be nice not to have to enter 3 every time you call this function. A closure would allow you to do this, by *closing* the value for *z* so that you didn't have to write it again and again. Such a closure would look like:

```
>>> def pow3(x, y):
...     return pow(x, y, 3)
...
```

Once defined, we can use our new function:

```
>>> pow3(3, 4)
0
>>> pow3(5, 4)
1
>>> pow3(5, 7)
2
```

Let's look at an example of a closure with a user-defined function:

```
>>> def f(x):
...     def g(y):
...         return x > y
...     return g
...
>>> lt5 = f(5)
>>> lt5(4)
True
>>> lt5(6)
False
>>>
```

Notice how we defined a function within an outer function, used a value from the outer function in the inner function, and then returned a reference to the just-defined inner function. This is a closure that encloses the value for *x*. Thus, when we use our new function to define the function *lt5()*, we get a custom function that tests for values less than 5. Similarly we could define *lt6*, *lt7*, etc.

Closures can often reduce the amount of code you have to write, if you need several similar functions that differ in only a few arguments. You might be wondering why, if closures are so useful, there is not a library function to make them easy to build. Well, there is! It is the function *partial* in the module *functools*:

```
>>> from functools import partial
>>> base2 = partial(int, base=2)
>>> base2('101010')          # Convert string in base 2 to an integer
42
>>>
```

The *partial* function works with any function that has keyword arguments. If you want to *close* (that is, freeze or lock in) positional arguments, they must be done in the order in which they occur in the function definition. For example:

```
>>> pow4 = partial(pow, 4)      # Function to raise 4 to any power
>>> pow4(2)                     # Compute 4 to the power of 2 (4 squared)
16
>>>
```

Naturally, you can use `partial` with user-defined functions as well as library functions.

3.4.4 Decorators

Sometimes you have a function, or a collection of similar functions, that do almost everything that you want. There's just one or two things you would like to add to reach your goal. This is where *decorators* come in. A decorator takes a function as an argument, adds something extra, then returns a new function that combines the original function with the extra abilities. Think about a birthday cake. You first bake the cake -- which is probably delicious on its own, then you make icing and decorate the cake with the icing. You might even go further and use a contrasting color icing to write "Happy Birthday" on the cake. That is just the kind of thing function decorators do. Let's look at a simple example:

```
>>> def decorator(f):
...     def _f(x):
...         print "I'm in f, calling", f, "with argument", x
...         return f(x)
...     return _f
...
>>> @decorator
... def g(x):
...     print "I'm in g with argument", x
...
>>> g(42)
I'm in f, calling <function g at 0x033D7970> with argument 2
I'm in g with argument 42
```

First, we defined a decorator function, which takes in just one argument, which is a function to be decorated. Within the decorator function, we define a second, nested function. (If you think this looks like a closure, you're right! Decorators are based on the same idea, but generally a decorator takes just one argument: a function reference.) The nested function simply reports which function it is calling and with what argument. Then, it calls the target function. Finally, the main decorator function returns the inner function as its result.

Having defined the decorator function, we use it in the definition of our next function. To to that, we use a special statement that begins with an '@' then continues with the name of the decorator function. In effect, we're putting icing on the cake! Finally, we define our *real* function. However, notice what happens when we call the real function. First, we get the line printed from the decorator function, then the line printed from the real function. This is the decorator in action.

Let's take a look at a more complicated example. This is also a great opportunity to introduce Python's exception-handling facilities. Say that you have a series of functions that read from a database. Each function takes in the name of the database and executes some special read operation. A skeleton of such a function might look like this:

```
def dbread1(dbname):
    """
    Do read type 1 from database dbname
    """
    data = dbname.read(...)    # Call the database read function
    return data
```

Now, suppose you want to be able to see errors that occur in the function `dbname.read()`. Suppose that that function may raise errors `DBERR1` and `DBERR2`. You want to report the errors to the user in some fashion for further analysis and do not want the user to just see cryptic error messages from the database itself.

You can define a decorator that you can use for all your `dbread()` functions that uses the Python `try...except` syntax to catch the exceptions. It might look like this:

```
def catch_errors(func):
    def _try(dbname):
        try:
            return func(dbname)
        except DBERR1, msg:
            print "Database error 1:", msg
            raise
        except DBERR2, msg:
            print "Database error 2:", msg
            raise
        except:
            print "Unknown database error occurred"
            raise
        else:
            # Database function completed normally
        finally:
            # Perform any needed cleanup
    return _try
```

The inner function, `_try()`, wraps the call to the database function in a `try...except` structure. First, the actual function call `func(dbname)` is attempted. Then, if either `DBERR1` or `DBERR2` occurs, the `except` clauses matching those two exceptions report on the error. If an unknown exception is encountered, the third `except` clause catches it (so it is the default error handler). If nothing goes wrong, the code in the `else` clause is executed. No matter what happens, the code in the `finally` clause is executed at the end (any cleanup code should go here).

Now that we have our decorator ready, we can use it to wrap any of our database functions and know that the errors will be caught and handled. We just have to write them like this:

```
@catch_errors
def dbread1(dbname):
    ...
@catch_errors
def dbread2(dbname):
```

...

.
. .
.

In case you're wondering, it is perfectly legal to stack decorators, which means this would be valid (assuming you have previously defined the decorator functions):

```
@happy_birthday
@icing
@top_layer
@filling
@bottom_layer
def plate(shape):
    ...
```

3.4.5 Co-routines and Consumers

If generators are data producers, co-routines are data consumers. Co-routines start off life looking a lot like generators. The difference is in how `yield` is used. Here's an example of a co-routine that consumes data:

```
>>> def co_routine(path):
...     f = open(path, 'w')
...     try:
...         while True:
...             data = yield          # yield expression
...             f.write(data + '\n')
...     except GeneratorExit:
...         f.close()
...
>>> f = co_routine("new.file")
>>> f.send(None)
>>> f.send('a')
>>> f.send('b')
>>> f.send('c')
>>> f.close()
>>>
>>> nf = open("new.file")
>>> for line in nf:
...     print line,
...
a
b
c
>>> nf.close()
```

(Admittedly, this co-routine does not do anything special. It just writes lines to a file specified when it is called. This could have been done without anything fancy, but it should serve to show how co-routines work

as data consumers and will hopefully inspire you to find more (and better!) uses for this approach.) The co-routine starts off like any other function, declaring its name and single argument. Then it opens the file to be written. Next, it enters a try...except construct, which begins with a while-loop. The loop is designed to run forever, or until an exception is raised. The single except statement looks for one exception: `GeneratorExit`.

In the main program code, the co-routine is called with the path name "new.file". Recall from our discussion about generators (which a co-routine is, since it contains a `yield`), that it won't start executing until it is asked for data, at which time it will advance to the first `yield` in the function. The next line is peculiar: `f.send(None)`. What this statement does, is that it causes the co-routine to start executing up to the first `yield`. Thus, the file is opened, the while-loop is entered and the function hits the first `yield`. Contrary to our generators, though, the `yield` is on the right-hand side of an assignment -- that is, it is actually an *expression*! More than that, it can consume data as well as produce it. So, when the `yield` is encountered for the first time, it first produces data (nothing, in this case), then -- if reached via the `send()` method -- waits for data from the caller.

The next three lines send three pieces of data to the co-routine, which completes the `yield` expression and assigns the result to the variable `data`, which is then written to the file. Finally, the statement `f.close()` tells the co-routine to stop, just like it would for a generator. Internally, it raises the exception `GeneratorExit`, which is caught by our `except` clause. The `except` clause just closes the file and the co-routine ends.

3.5 Lambda Expressions

Lambda expressions are a way of creating anonymous functions. Here's a simple one:

```
>>> lambda x:x*x
```

If you guessed that that simply squares a given number, you'd be correct. In fact, we can take this simple expression and assign it to some variable:

```
>>> square = lambda x:x*x
```

which is exactly the same as defining a `square` function like this:

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

"So what?" you may ask. I save a little on typing, big deal! Well, there are great uses for lambda expressions. For example, some functions take functions as arguments. Remember the `partial()` function? However, using a lambda with a `partial` is no easier than defining the function in the first place!

One of the methods you can use with Python lists is the `sort()`. This method has an optional first argument that is -- you guessed it! -- a function. So I can write:

```
>>> mylist.sort(lambda x,y: cmp(y, x))
```

`cmp` is a built-in function that compares two objects. Here we use a lambda in place of a previously-defined function to perhaps change the sort order based on some rule of our own. (In the example, the rule would cause the list to be sorted in descending order.)

Lambdas also have great applications when writing GUI programs where things happen asynchronously. (This is a large topic that is out of scope for this introductory volume.)

Lambdas are a great tool to have in your kit!

3.6 Exercises

1. Fibonacci numbers in constant time

(a) The standard definition of the Fibonacci sequence is recursive:

$$F_n = F_{n-1} + F_{n-2}$$

It can be shown, however, that there is a closed (i.e. non-recursive) formula (Binet's formula, after French mathematician Jacques Philippe Marie Binet, 1786-1856) for the sequence that is based on the "golden-ratio." Using that formula and doing a little algebra, it can be reduced to this:

$$F_n = \text{floor} \left(\frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right), \text{ for all } n > 0$$

where φ is the golden ratio:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Write a function that computes the n'th Fibonacci number using this formula. Note that your function should have no loops in it-and from that perspective it can be said to have a constant complexity or $O(1)$ using "big O" notation.

(b) There is a module in the standard library called `timeit` that can be used to time snippets of code. You can use it like this:

```
>>> from timeit import Timer
>>> def fibonacci(n):
...     """
...     Computes the n'th Fibonacci number in linear time.
...
...     Input argument: n - an integer
...     Returns: the n'th Fibonacci number
...     """
...
...     if n == 0:
...         return 0
...     fc = fn = 1
...     for _ in xrange(2, n):
...         fn, fc = fn + fc, fn
...     return fn
...
>>> Timer('fibonacci(10)', 'from __main__ import fibonacci').timeit()
```

```
2.3244328498840332
>>>
```

By default, the `timeit()` method runs the sample code 1,000,000 (yes, one *million*) times and reports on the total time taken in seconds. For this exercise, use the `timeit` module to time your new function from part (a) and compare it to the times for the same Fibonacci numbers using the above function. As the Fibonacci numbers increase, which implementation (the given one or your new "constant-time" version), produces the fastest result? Do they both produce the same results for all values of `n` up to 100? If not, why not?

2. Decorator to help set-up a co-routine and automatically advance to first yield.

Recall that a co-routine built using `yield` expressions does not start until the first `next()` or `send(None)` call is received. However, it is all too easy to forget to do that setup call before using the co-routine. Write a function decorator that can be used to automate the first call to advance the co-routine to the first `yield` statement. Demonstrate the use of your decorator on the co-routine from section 3.4.5.

3. Write a generator to list all Python files in the standard library

Recall that the `sys` module has a `path` attribute that is a list of all paths that Python will search to find modules when processing an import statement. Depending on your operating system, the list may look something like this:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2',
'/usr/lib/python2.6/lib-tk', ...]
>>>
```

Note the list item that contains the path to the beginning of the Python library. (In the above example, it is `sys.path[1]`).

In the `OS` module there is a function named `walk` that traverses the entire tree under a given starting directory and returns a tuple for each one that contains three items: the directory path, the subdirectories contained in that directory and the file names contained in that directory.

For this exercise, write a generator that uses these modules to generate file names in the Python standard library where the last three characters are `".py"`. (Hint, all Python strings have a method called `endswith()` that can be used to check if the string ends with some sequence of characters.)

4. List Maintenance

(a) There is a library module named `bisect` that can be used to perform binary searches on sorted lists to quickly find items in the list. For example, if you have a Python list called `mylist` that is already sorted, you can find an item in the list like this:

```
>>> from bisect import bisect_left
```

```
>>> mylist = ['abc','def','ghi','klm','opq','rst','uvw','xyz']
>>> bisect_left(mylist, 'klm')
3
>>> mylist[3]
'klm'
>>>
```

Note that, for larger, sorted lists, bisection is substantially faster than the list method `index()`, which searches the list sequentially from the beginning.

Write a function that uses bisection. The function must take three arguments: a list, an item and an operation. The first argument can be any valid Python list (make sure that it is!); the second argument can be any item that might be in the list; the third parameter is an integer code specifying an operation:

Code	Operation
0	Sort the list. Returns None.
1	Insert item into list using bisection. Returns True if the new item is already in the list (but inserts a duplicate anyway), otherwise returns False.
2	Remove item from list using bisection. Returns True if item found, otherwise returns False.
4	Returns True if item is in the list, otherwise return False

Be sure to test your function on a wide variety of lists and items and operations, including empty lists and null items. (Note that None is a valid value for a list item but you have to decide whether you want to include it or not.)

(b) Write four closures for your function to make the four operations usable without specifying the third argument.

(c) Suppose that your program only operates on a single, large list. Use the partial function to create closures to call your new function so that you can omit the first argument.

4

Chapter 4 – Sequences, Mappings and Collections

So far we have considered a limited subset of Python data types: integers, floating point numbers, strings and lists. In this chapter we will look at some of the other main Python data types and some of the extensions built on top of them. These types fall into the categories of Sequences, Collections and Mappings.

4.1 Sequences

Sequences are objects over which it is possible to iterate. That is, for every sequence type, this must work:

```
for item in sequence:
    # do something with item
```

In Chapter 3 we looked at lists, which are one of the most commonly-used Python sequences. We saw how to build lists, add and remove items, slice them into smaller lists and iterate over them. However, back in Chapter 2, we introduced one common sequence type, “string”. Now we will discuss it in further detail.

4.2 More fun with strings

4.2.1 Multi-line strings

So far, we know how to build strings. We simply enclose some characters in quotation marks. We also learnt that there are four ways to do this:

- Single quotation marks, e.g. 'string in single quotation marks'
- Double quotation marks, e.g. "string in double quotation marks"
- Triple-single quotation marks, e.g. """string in triple single quotation marks"""
- Triple-double quotation marks, e.g. """string in triple double quotation marks"""

We also learned that triple-double quotation marks are used to document functions (the docstring). However the two "triple" forms have a more general purpose: multi-line strings. They are very useful. Suppose you wanted to write three lines to a file. If you used either of the first-two forms, you might do something like this:

```
>>> line1 = "first line"           # String for the first line
>>> line2 = "second line"          # String for the second line
>>> line3 = "third line"           # String for the third line
```

```
>>> f = open("/tmp/3lines.txt", 'w') # Open a file to write
>>> f.write(line1 + '\n')             # Write first line, appending a new line
>>> f.write(line2 + '\n')             # Write second line, appending a new line
>>> f.write(line3 + '\n')             # Write third line, appending a new line
>>> f.close()                         # Close the file
>>>
```

What if you then had to go back and add another three lines? You'd have to also add three new write() calls. If your output file grew and grew, then the maintenance would become boring at least, and likely be error-prone as well. Wouldn't it be nice if you could define the content in one place and then write it in one operation? Well, if you use a multi-line string, you can do just that:

```
>>> lines = """first line
... second line
... third line
... """
>>> f = open("/tmp/3lines.txt", 'w')
>>> f.write(lines)
>>> f.close()
>>>
```

A multi-line string, like the one above, can be defined in one place and maintained on its own. The code that actually writes the data need not change if the multi-line string changes. Also, a multi-line string automatically adds new line commands at the end of each line, so that we do not need to add them by hand.

A multi-line string can also be a handy way to write data that requires substitution and can be performed with the format() method from Chapter 3:

```
>>> HTML = """
... <html>
... <head>
...     <title>{title}</title>
... </head>
... </body>
... This is easy to do in {lang}
... using {how}.
... </body>
... </html>
... """
>>> f = open("/tmp/multi.html", 'w')
>>> f.write(
...     HTML.format(title="Fun with strings", lang="Python", how="multi-line
... strings")
... )
>>> f.close()
>>>
```

Here, we have formatted a simple page of HTML, using a multi-line string to define a template with substitution variables, then using the format() method on it to substitute for the three variables. Note that we

used all capitals for the identifier for the multi-line string constant. This is a convention used in Python programs for defining constants and is part of PEP8.

Constants are usually declared on a module level and written in all capital letters with underscores separating words. Examples include MAX_OVERFLOW and TOTAL.

This approach can be very handy for dynamically generating text that is actually code in another language. Using a multi-line string and formatting, it is easy to generate a segment of JavaScript, XML, SQL or any other language at execution time.

4.2.2 Other String Methods

Strings have a number of very useful methods that you can use for querying and modifying strings. Also, some of the arithmetic operators take on new meaning with strings. The table below shows some of these and give typical use cases for them. For a full list, see the official documentation.

Method or operator	Purpose	Example
+	Concatenation	<pre>>>> 'a' + 'b' 'ab'</pre>
*	Multiplication	<pre>>>> 'a' * 5 'aaaaa'</pre>
[not] in	Return True if string1 [not] found in string2	<pre>>>> 'a' in 'bac' True</pre>
[index or slice]	Return string or slice	<pre>>>> a = 'abcd' >>> a[1] 'b' >>> a[2:4] 'cd' >>> a = 'abcd' >>> a[-1] 'd'</pre>
len()	Return length of string	<pre>>>> len("abcd") 4</pre>
capitalize()	Return a new string with the first letter capitalized	<pre>>>> "the string".capitalize() 'The string'</pre>
center(width, [pad])	Return string centered in a new string of length width padded with pad (default blank)	<pre>>>> "string".center(10) ' string ' >>></pre>
count(sub[, start[, end]])	Returns number of non-overlapping occurrences of sub in the range [start, end]	<pre>>>> "aaabbcc".count('a') 3</pre>

<code>find(sub[, start[, end]])</code>	Returns index of start of <i>sub</i> in range [<i>start</i> , <i>end</i>] or -1 if not found	>>> "aaabbcc".find("bc") 4
<code>lower()</code>	Returns a copy of the string translated to lower case	>>> "HTML".lower() 'html'
<code>partition(sep)</code>	Splits the string at <i>sep</i> and returns a 3-tuple with the first part, the separator and the last part.	>>> 'Guido van Rossum'.partition('van') ('Guido ', 'van', ' Rossum')
<code>split([sep[, maxsplit]])</code>	Splits the string using <i>sep</i> as a separator and returns a list of words. Limits the split by <i>maxsplit</i> .	>>> sys.path[1].split('/') ['', 'usr', 'lib', 'python2.6']
<code>strip([chars])</code>	Returns a new string with leading and trailing characters removed. <i>chars</i> defaults to a space	>>> " too wide" .strip() 'too wide'

4.2.3 More on lists

Back in Chapter 3 we introduced lists and tuples. There is much more to say about these useful data types!

4.2.4 Updating and deleting list items

It's easy to update and delete items in a list:

```
>>> del mylist[1]           # remove the second item from the list

>>> mylist[1] = "was"      # change the value of the second item to "was"

>>> del mylist[:]          # empty the list
```

In the last example, note that if the beginning and/or end of a slice is not specified, a slice begins at item 0, the first item, and ends after the last item. Thus, a slice of `[:]` refers to the whole list, a slice of `[1:]` means the slice from the second item to the end and a slice `[:-1]` means everything but the last item.

4.2.5 Operators and methods for lists and tuples

Python lists and tuples come with several useful methods, which we list here:

Method	Purpose	Example
+	Add items from another list	>>> [1, 2, 3] + [4, 5, 6] [1, 2, 3, 4, 5, 6]
*	Duplicate list items	>>> ['foo']*3 ['foo', 'foo', 'foo']
append(x)	Add item x to the end of the list	mylist.append("example")
count(x)	Count the number of items matching x	mylist.count("list")
extend(x)	Extend the list by adding the items from the list x to the end	mylist.extend(["more", "items"])
index(x)	Return the index number of item x, if found in the list	mylist.index("list")
pop(x)	Remove and return item at index x (default last)	mylist.pop(0) # remove and return item 0
remove(x)	Remove the first occurrence of item with the value x	mylist.remove("list")
reverse()	Reverse the order of the items in the list	mylist.reverse()
sort(key=None, reverse=False)	Sort the items in the list, optionally using the key function "key" and/or in reverse order	mylist.sort()

Since tuples are immutable, the only methods available are count() and index(). Both operators are available for tuples. In addition, since it is immutable, a new tuple will be created, rather than changing the current one.

4.3 Sets

4.3.1 Introduction to Python Sets

Sets in Python are used to implement the mathematical concept of a set. That is, they are unordered collections of immutable objects. Since they are unordered, they cannot be indexed. However it is possible

to iterate over them. To build a set, we use the `set()` built-in function, which takes an iterable as its single, optional argument. For example:

```
>>> myset = set([1,2,3,4,5,4,3,2,1])
>>> myset
set([1, 2, 3, 4, 5])
>>> myset2 = {7,8,9}      # short-hand set definition
>>> myset2
set([8, 9, 7])
```

Notice that the set contains only unique objects. The extra list entries that we supplied ([4,3,2,1]) were not added to the set since they were already present. Also notice that there is no order to members of a set.

4.3.2 Set operations

Sets support the typical operations of union, intersection and difference, using the operators `|`, `&` and `-`, respectively. Also available is the symmetric difference operator `^`, which returns a new set with elements that are in one set or the other, but not both. For example:

```
>>> myset2 = {i for i in xrange(4,8)}      # use a comprehension
>>> myset
set([1, 2, 3, 4, 5])
>>> myset | myset2                        # Union
set([1, 2, 3, 4, 5, 6, 7])
>>> myset & myset2                        # Intersection
set([4, 5])
>>> myset - myset2                        # Difference
set([1, 2, 3])
>>> myset ^ myset2                        # Symmetric difference
set([1, 2, 3, 6, 7])
```

4.3.3 Set Testing

Also available are the typical Boolean set operations for testing set memberships and the (proper) subset/superset operations:

```
>>> myset
set([1, 2, 3, 4, 5])
>>> myset2
set([4, 5, 6, 7])
>>> myset3 = set([1,2,3])
>>> 1 in myset                          # Set membership
True
>>> 6 in myset                          # Set membership
False
>>> 4 not in myset3                     # Set membership
True
>>> myset3 < myset                      # Proper subset
```

```

True
>>> myset < myset          # Proper subset
False
>>> myset3 <= myset         # Subset
True
>>> myset > myset3          # Proper superset
True
>>> mset >= myset3          # Superset
True
>>> myset >= myset2         # Superset
False
>>>

```

We summarize these operators in this table:

Set operation	Python operator
\cup (union)	
\cap (intersection)	&
$-$ (difference)	-
Δ (symmetric difference)	^
\in (membership)	in
\notin (non – membership)	not in
\subset (proper subset)	<
\subseteq (subset)	<=
\supset (proper superset)	>
\supseteq (superset)	>=
$ x $ (cardinality)	len (x)

4.3.4 Sets as relations

Sets are handy for quick testing of memberships and for removing duplicates from a list or tuple or any other iterables. Since members of a set can also be tuples, they can be used to represent relations as in a relational database and even perform relation algebra or SQL-like queries. For example, let's generate a set of tuples using a library function that generates random numbers:

```
>>> from random import randint
>>> R = set(
... (randint(1,10), randint(1, 10), randint(1,10))
... for _ in range(100)
... )
```

Notice that we used a comprehension in the call to `set()`. Such comprehensions can be used wherever iterables are called for. It is of course possible to build `R` without using a comprehension. The code would look something like this:

```
>>> R = set()
>>> for _ in range(100):
...     R.add(
...         (randint(1,10), randint(1,10), randint(1,10))
...     )
...
>>>
```

This approach will generally be slower though, since it has separate calls to the `add()` method for each iteration. Use the `timeit` module to prove that to yourself.

Now, let's query our relation "R". Assume that it has the attribute columns (A, B, C).

```
>>> # SELECT A, B FROM R WHERE A = 1
>>> for t in set((A, B) for (A, B, C) in R if C == 1): print t,
...
(5, 4) (10, 4) (4, 5) (6, 1) (3, 1) (9, 10) (1, 5) (8, 8) (2, 2) (3, 7) (1, 1)
(6, 5)
```

Notice how Python used the names 'A', 'B', and 'C' in the statement to extract the members of the tuples. This is a very pythonic way to process tuples and lists and other collections.

For more examples of SQL-like relations, see the section on named tuples, below.

4.3.5 Set Methods

Like other built-in objects in Python, sets have an assortment of methods for processing, in addition to those that have been discussed already. For the examples, we assume that the two sets, `R` and `S` are defined as follows:

```
>>> R = set([1, 2, 3])
```

```
>>> S = set([4, 5, 6])
```

Method	Purpose	Example
<code>isdisjoint(<i>other</i>)</code>	Returns True if the set and <i>other</i> have no elements in common.	<pre>>>> R.isdisjoint(S) True</pre>
<code>union(<i>other</i>, ...)</code>	Returns a new set from the union of the set and all <i>others</i> .	<pre>>>> R.union(S, [7, 8, 9]) set([1, 2, 3, 4, 5, 6, 7, 8, 9])</pre>
<code>intersection(<i>other</i>, ...)</code>	Returns a new set from the intersection of the set and all <i>others</i> .	<pre>>>> R.intersection(S) set([])</pre>
<code>difference(<i>other</i>, ...)</code>	Returns a new set from the difference of the set and all <i>others</i> .	<pre>>>> R.difference(S) set([1, 2, 3])</pre>
<code>symmetric_difference(<i>other</i>, ...)</code>	Returns a new set from the symmetric difference of the set and all <i>others</i> .	<pre>>>> R.symmetric_difference(S) set([1, 2, 3, 4, 5, 6])</pre>
<code>add(<i>elem</i>)</code>	Adds element <i>elem</i> to the set.	<pre>>>> R.add(0) >>> R set([0, 1, 2, 3])</pre>
<code>update(<i>other</i>, ...)</code>	Updates the set, adding elements from <i>others</i> .	<pre>>>> R.update([-1, -2]) >>> R set([0, 1, 2, 3, -1, -2])</pre>
<code>remove(<i>elem</i>)</code>	Removes an element <i>elem</i> from the set. Raises <code>KeyError</code> if <i>elem</i> is not in set.	<pre>>>> R.remove(3) >>> R set([0, 1, 2, -1, -2])</pre>
<code>discard(<i>elem</i>)</code>	Discards the element <i>elem</i> from the set if it is present.	<pre>>>> R.discard(3) >>> R set([0, 1, 2, -1, -2])</pre>
<code>pop()</code>	Returns and remove an arbitrary element from the set.	<pre>>>> R.pop() 0</pre>
<code>clear()</code>	Removes all of the elements from the set.	<pre>>>> R.clear() >>> R set([]) >>></pre>
<code>len()</code>	Returns the set cardinality.	<pre>>>> len(S) 3</pre>

4.3.6 Immutable Sets

The standard `set()` object has an immutable cousin: `frozenset()`. A frozen set is a set that you cannot modify once it is built. Therefore, it has none of the methods described above that can be used to add, remove or update items in the set. You can build a frozen set from another set or other iterable (including a comprehension):

```
>>> from random import randint
>>> R = frozenset(
...     (randint(1,10), randint(1, 10), randint(1,10))
...     for _ in range(100)
... )
```

However, you cannot build a frozen set using the alternate, for-loop method:

```
>>> R = frozenset()
>>> for _ in range(100):
...     R.add(
...         (randint(1,10), randint(1,10), randint(1,10))
...     )
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Frozen sets are useful for testing fixed memberships. For example, suppose you had a list of all the battleships that were sunk in WWII. If, at the start of your program, you put these into a frozen set, then you can test any ship name to see that was among those that had sunk:

```
>>> sunken_battleships = frozenset(...)
>>> if shipname in sunken_battleships:
>>>     print "The battleship {0} was sunk during World War
II".format(shipname)
```

4.4 Dictionaries

4.4.1 Building dictionaries

Dictionaries are a simple way to map unique keys to data. For example, suppose you want to keep track of phone numbers. You could use a Python dictionary, keyed by friends' names, where the data portion is a phone number:

```
>>> number = dict(Bob="212-555-1212", Ted="214-555-1212", Carol="416-555-1212",
Alice="514-555-1212")
>>> number["Bob"]
'212-555-1212'
```

Here, we built a simple dictionary and added four telephone numbers to it. (Your friends may be good sources of information!). We then retrieved Bob's telephone number using the standard look up. Python also provides a simplified syntax for defining dictionaries with static values:

```
>>> d = {"Bob": "212-555-1212", "Ted": "214-555-1212", "Carol": "416-555-1212", "Alice": "514-555-1212"}
```

This produces the same dictionary as in the first example. Between the braces (`{...}`) you can use keywords, as above, or two-member lists or tuples. See the official documentation for a full description.

4.4.2 Building dictionaries with list comprehensions

Here's another dictionary containing the first 100 squares and that was built using a comprehension:

```
>>> one100squares = {k:k**2 for k in range(100)}
>>> one100squares.get(99)
9801
>>> one100squares[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
>>> one100squares.get(100)
>>> one100squares.get(100, "Key not in dictionary")
'Key not in dictionary'
```

In this example, the dictionary is built from the successive items that are generated by the comprehension. The first is `[1, 1]`; the second `[2, 4]` and so on up to `[99, 9801]`. Notice how the `get()` method operates. If we ask for a key that is not there, we get nothing (and no exception is raised). However, we can supply a second argument to the `get()` function, which is a default value that is to be returned if the key is not in the dictionary.

4.4.2 Modifying Dictionary Entries

We can easily modify or remove entries in a dictionary:

```
>>> one100squares[100] = 100**2      # Set up the entry for one hundred squared
>>> one100squares[100]
10000
>>> one100squares[100] = 100**2.1    # Change the entry to an illogical value
>>> one100squares[100]
15848.931924611141
>>> one100squares.setdefault(100, 100**3) # Try to set an entry to 100 cubed
15848.931924611141
>>> one100squares.setdefault(101, 101**2) # Try to set 101 cubed
10201
>>>
```

```
>>> del one100squares[100]          # Remove entry with the key equal to 100
>>> one100squares[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
>>>
```

Notice that the `setdefault()` method does *not* modify the data associated with a key already in the dictionary. However, if there is no such key, it adds the key with the data portion set to the second argument. Also, `setdefault()` returns the current value at that key after it is done.

For bulk updates to dictionaries, there is an `update()` method that takes another dictionary or list or tuple of key/value pairs as an argument and updates the dictionary with those new or changed values. For example:

```
>>> d = dict( (k, k**2) for k in range(5))
>>> d
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> d.update((k, k**3) for k in range(3,7))
>>> d
{0: 0, 1: 1, 2: 4, 3: 27, 4: 64, 5: 125, 6: 216}
>>>
```

4.4.3 Dictionaries as Keyed Relations

Dictionary keys and elements do not have to be simple strings and numbers. They can be many things (even other dictionaries). You could use a dictionary to emulate a keyed table in an RDBMS:

```
>>> from random import randint
>>> R = set(
... (randint(1,10), randint(1, 10), randint(1,10))
... for _ in range(100)
... )
>>>
>>> keyed_R = dict(
... (A, (A, B, C))
... for A, B, C in R
... if C == 1
... )
>>> keyed_R[10]
(10, 4, 1)
```

Here, we've emulated a table with the attributes (A, B, C) using A as the primary key.

4.4.4 Iterating Over a Dictionary

Python provides four ways to iterate over dictionaries. You can do that by key values, by data values or by both at the same time. Here are some examples:

```
>>> d = dict( (k, k**2) for k in range(3))
>>> d
{0: 0, 1: 1, 2: 4}
>>> for k in d:                                # Iterate over the keys (the default)
...     print k
...
0
1
2
>>> for k in d.iterkeys():                      # Iterate over the keys (explicitly)
...     print k
...
0
1
2
>>> for v in d.itervalues():                   # Iterate over the values
...     print v
...
0
1
4
>>> for i in d.iteritems():                   # Iterate over the key/value pairs as tuples
...     print i
...
(0, 0)
(1, 1)
(2, 4)
>>>
>>> for k, v in d.iteritems(): # Iterate over key, value pairs
...     print k, v
...
0 0
1 1
2 4
>>>
```

The last example shows that, when the items returned by an iterator are tuples (or lists), you can "take them apart" but specifying multiple loop variables, separated by commas.

There is also a method that allows you to extract key/value pairs from a dictionary until it is empty. This method is called `popitem()`. Here it is in action:

```
>>> while len(d):
...     print d.popitem()
...
(0, 0)
(1, 1)
(2, 4)
>>>
```

4.4.5 Dictionary Methods

The following table lists the operators and methods that can be used on dictionaries:

Operator or Method	Purpose	Example
<i>key</i> [not] in d	Returns True if <i>key</i> is [not] in the dictionary.	>>> 1 in {1:1} True
len(d)	Returns the number of items in the dictionary.	>>> len({1:1}) 1
d[key]	Returns the value of item having <i>key</i> as the key.	>>> {1:2}[1] 2
d[key] = <i>value</i>	Creates or updates the item having <i>key</i> as the key to the value <i>value</i> .	>>> d = {1:2} >>> d[1] = 3
del d[key]	Removes the item with the key <i>key</i> from dictionary.	>>> del d[1]
clear()	Removes all items from the dictionary.	>>> d.clear()
copy() dict(d)	Returns a copy of the dictionary.	>>> d.copy() >>> dict(d)
fromkeys(<i>seq</i> , <i>value</i>)	Creates a new dictionary with the keys from <i>seq</i> and the values set to <i>value</i> .	>>> dict().fromkeys(['a','b'],0) {'a': 0, 'b': 0}
get(<i>key</i> [, <i>default</i>])	Returns the item with key <i>key</i> from the dictionary, or the <i>default</i> value if <i>key</i> not found.	>>> d.get('a', "Not Found")
items()	Returns a list of the dictionary's (key, value) pairs.	>>> {1:1, 2:2}.items() [(1, 3), (2, 4)]
keys()	Returns a list of the dictionary's keys.	>>> {1:1, 2:2}.keys() [1, 2]
values()	Returns a list of the dictionary's values.	>>> {1:3, 2:4}.values() [3, 4]

<code>iteritems()</code>	Returns an iterator over a dictionary's items.	<pre>>>> for k, v in d.iteritems():</pre>
<code>iterkeys()</code>	Returns an iterator over a dictionary's keys.	<pre>>>> for k in d.iterkeys():</pre>
<code>itervalues()</code>	Returns an iterator over a dictionary's values	<pre>>>> for v in d.itervalues():</pre>
<code>pop(key[, default])</code>	Removes the item with the key <i>key</i> from dictionary and return its value or <i>default</i> if not found.	<pre>>>> d.pop('a', "Not found")</pre>
<code>popitem()</code>	Removes and returns an arbitrary (key, value) pair from the dictionary.	<pre>>>> {1:3, 2:4}.popitem() (1, 3)</pre>
<code>setdefault(key[, default])</code>	If <i>key</i> is in the dictionary, return its value. Otherwise, insert a new item with the key <i>key</i> and value <i>default</i> and return <i>default</i> .	<pre>>>> {1:3, 2:4}.setdefault(1, 4) 3 >>> {1:3, 2:4}.setdefault(3, 5) 5</pre>
<code>update(other)</code>	Update the dictionary from <i>other</i> iterable or keyword arguments	<pre>>>> d = {1:2, 3:4} >>> d.update(a=5) >>> d {'a': 5, 1: 2, 3: 4} >>> d.update({6:7}) >>> d {'a': 5, 1: 2, 3: 4, 6: 7}</pre>

4.4.6 Using Dictionaries to Categorize and Count

One useful application for dictionaries is categorizing and counting. Suppose your program reads movie data, where each movie is represented by a list, and where the first element is the movie title, the second element is the director of the movie and the other elements include other data. Suppose you want to count the number of movies produced by each directory. Using standard dictionary methods, you might do something like this:

```
>>> directors = {}
>>> for movie in get_movies():
...     if movie[1] in directors:
...         directors[movie[1]] += 1
...     else:
...         directors[movie[1]] = 1
```

Alternatively, you could use the `get()` method described above to simplify your program:

```
>>> directors = {}
>>> for movie in get_movies():
...     directors[movie[1]] = directors.get(movie[1], 0) + 1
```

4.4.7 Using defaultdict

There is, however, a special kind of dictionary available in the collections module called defaultdict: Here is how we can use it to solve the same problem:

```
>>> from collections import defaultdict
>>> directors = defaultdict(int)
>>> for movie in get_movies():
...     directors[movie[1]] += 1
```

A defaultdict sets a default data type to be used to automatically create entries for keys that are not in the dictionary. In this example, we used the data type `int` which means that for every new key, an entry will be added to the dictionary with the key `movies[1]` and the value `int()`, which defaults to 0. Note that the argument supplied to `defaultdict` must be *callable* -- that is it must be some sort of function or method.

That is why you cannot just code `defaultdict(0)`, even though at first glance, that might seem to make sense. In the example below, the first time Stephen Spielberg is encountered, an entry will be added with his name as the key and a data value of 0. However, this happens inside an augmented assignment statement. Thus, the very next thing that happens is that 1 is added to the entry. $0 + 1 = 1$ so the entry now has Stephen Spielberg as the key and 1 as the value, as shown below:

```
>>> from collections import defaultdict
>>> directors = defaultdict(int)
>>> directors["Stephen Spielberg"] += 1
>>> for k, v in directors.iteritems():
...     print k, v
...
Stephen Spielberg 1
```

Suppose that, instead of counting movies, you want to make a list of the movie data by director. `defaultdict` can handle this task as well:

```
>>> from collections import defaultdict
>>> movies_by_director = defaultdict(list)
>>> for movie in get_movies():
...     movies_by_director[movie[1]] += [
...         movie[0:0] + movie[2:]
...     ]
```

The entries of the `movies_by_director` dictionary will consist of a list of movie data. Every item in the list will itself be a list of data extracted from the items returned by the function `get_movies()`, excluding the director's name (which becomes the key).

4.5 Named Tuples

Sometimes, the lists and tuples have a structure that is not visible at first. For example, if we read data from a database using an SQL query, we will receive a list of items for each row of the query. However, the list

contains the attributes specified in the SQL query command and in the order specified. That means that if I send this command to an RDBMS:

```
SELECT A, B, C FROM R;
```

We will get back rows with the attributes in that order. However, this data is returned to a Python program as a simple list. If we have a Python variable called `row`, for example, we have to remember that `row[0]` is "A", `row[1]` is "B" and `row[2]` is "C". Not only is this tedious, it is error-prone, since I might inadvertently code the wrong item number. Worse, what if the `SELECT` statement is modified at some point to return the attributes in a different order? Named tuples give us a method to avoid these problem and make our programs more readable and maintainable as well. Let's look at an example of how we might do that:

```
>>> from collections import namedtuple # Get the namedtuple type
>>> R = namedtuple("R", "A, B, C")      # Define a named tuple for our relation
>>> conn = connect("database")          # Connect to our database
>>> cursor = conn.cursor()              # Get a cursor
>>> cursor.execute("SELECT A, B, C from R") # Execute an SQL SELECT
>>> for row in map(R._make, cursor.fetchall()):
...     # Iterate over the rows that we got back
...     print R.A, R.B, R.C
...     # Print the attributes from each row
```

Taking this apart, we first import the module we need, then use the `namedtuple` function to set up names for our relation "R". In the next three lines, we connect to the database. (We'll cover database access methods in a later chapter). Our for loop uses the named tuple. Let's look at it more closely. First of all, it uses the `map` function, which takes at least two operands. The first is a function to be performed -- `R._make` in this case; the second is an iterator that returns 0 or more items -- `cursor.fetchall()` in this case. Then, `map` calls the function for every item returned by the iterator and returns the result to the caller, the for-loop in this case.

What does the function `R._make` do? This function was created in the second statement of the example that called `namedtuple`. The `_make` function then takes whatever it gets and returns a structure so that you can refer to the items by name. Here is another example with some real data:

```
>>> movies = [
...     ["Star Wars", "George Lucas"],
...     ["The Matrix", "Andy and Lana Wachowski"],
...     ["Close Encounters", "Stephen Spielberg"],
... ]
>>> movie = namedtuple("Movies", "title, director")
>>> for mov in map(movie._make, movies):
...     print "{0:<20} {1:<20}".format(mov.title, mov.director)
...
Star Wars           George Lucas
The Matrix          Andy and Lana Wachowski
Close Encounters    Stephen Spielberg
```

If, at some future point, the data is presented in a different order, the only thing we have to change is the call to `namedtuple`. The rest of our program can continue as unchanged.

4.6 Double-ended Queues

A common programming problem is to manage lists of things to do or things to work on. Sometimes you will want to put things on the end of the list and take them off the start (FIFO, or first in, first out); other times you will need to put them on the start and take them from the start (LIFO, or last in, first out). You would be able to do this using regular Python lists and the `append()` and `pop()` methods, but these operations are so common that there is a special, high-performance data type customized for just this sort of use. It is called a `deque`, which is pronounced like "deck" and stands for "double-ended queue". This data type is found in the `collections` module, where `namedtuple` is also found. Here is a simple example:

```
>>> from collections import deque          # Import the deque data type
>>> lifo = deque()                         # Make a new deque
>>> for i in range(5):                     # Add 5 new items
...     lifo.append(i)
...
>>> while lifo:                            # Take off all items, LIFO
...     print lifo.pop()
...
4
3
2
1
0
```

Of course you can put anything you like into a deque: integers, strings, lists, tuples, dictionaries, functions -- even other deques! Here's how you could use a deque for a FIFO queue:

```
>>> fifo = deque(i for i in range(5))      # Make a FIFO deque
>>> while fifo:
...     print(fifo.popleft())              # Take off all items, FIFO
...
0
1
2
3
4
```

Here, we show that using the `popleft()` method removes items from the left of the queue. We can also see that a `deque` can take a comprehension (or other iterable) as an argument.

The following table lists the methods that can be used with deques. Assume that we have a deque called

"deck".

Method	Purpose	Example
<code>deque(iterable, maxlen)</code>	Make a new deque from <i>iterable</i> , limiting its length to <i>maxlen</i> (default is no limit).	<code>deck = deque(maxlen=10)</code>
<code>append(item)</code>	Add an item to the right side of the deque.	<code>deck.append(1)</code>
<code>appendleft(item)</code>	Add an item to the left side of the deque.	<code>deck.appendleft("Deques are fun!")</code>
<code>clear()</code>	Removes all of the items from the deque.	<code>deck.clear()</code>
<code>extend(iterable)</code>	Adds all the items from <i>iterable</i> to the right side of the deque.	<code>deck.append([1, 2, 3])</code>
<code>extendleft(iterable)</code>	Add all the items from <i>iterable</i> to the left side of the deque.	<code>deck.extendleft(range(10, 5, -1))</code>
<code>len(deque)</code>	Returns the number of items in the <i>deque</i> .	<code>len(deck)</code>
<code>pop()</code>	Remove and return the item from the right side of deque.	<code>deck.pop()</code>
<code>popleft()</code>	Remove and return the item from the left side of deque.	<code>deck.popleft()</code>
<code>remove(value)</code>	Remove the first item found (from left) that is equal to <i>value</i> .	<code>deck.remove(1)</code>
<code>rotate(n)</code>	Rotate the deque <i>n</i> items to the right. If <i>n</i> is negative, rotate left instead.	<code>deck.rotate(1)</code>

4.7 Exercises

1. Using a Dictionary to Collect Objects

Write a function that examines Python's search path and builds and returns a dictionary that collects the paths by the prefix consisting of the first two directories in the path. All of the directories below that, will be in a list that is the data portion of the dictionary entry. For example, if the search path had these entries:

```
/foo/bar/fum
/foo/bar/foe
/fum/foo/bar
```

Your dictionary would look like:

```
{"foo/bar": ["fum", "foe"], "fum/foo": ["bar"]}
```

Be sure to allow for empty strings in the path and for path prefixes with no subdirectories. To test your function, write a second function that takes the dictionary you build and prints it by each entry in the dictionary, indenting the subdirectories four spaces. With the above example, your output should look like:

```
/foo/bar
    fum
    foe
/fum/foo
    bar
```

2. Using a dictionary like a keyed table in a relational database

Assume that you have the following dictionary structure for the movie data:

```
{"title year": ("director", ("star 1", "star 2", ...)), ...}
```

One entry might be:

```
{"Star Wars 1977": ("George Lucas", ("Carrie Fisher", "Mark Hamill", "Harrison Ford"))}
```

(a) Write a function that builds the dictionary by prompting the user for input for each piece of data. (You'll need a way to indicate that you have entered the last star name.) Populate the dictionary using your function and data from movies that you like.

(b) Write a function that accepts the dictionary you built in part (a) and a second argument with the last name of a star. Print all movies where that star appears in your dictionary. Format the print using indentation and labels for the title, year, director and cast.

3. Using a list like a matrix

A Python list can have lists as entries. That means that it is not difficult to define arrays, since a two-dimensional array is no more than a table with rows (the outer list) and columns in each row (the inner list).

(a) Write a function that builds an array using such a two-dimensional list. The function should take as input the number of rows and columns for the array and populate the array with random integers in a range specified as the third and fourth arguments of the function. A typical function call would look like:

```
myarray = build_array(10,10,1,100)
```

(b) Write a second function that performs matrix multiplication. This function will take in two arrays and return the result of the computation. It must check that the matrices are fully compatible for the operation. Using the result from part (a), test this function and check your results by hand or by any other means that you like.

5

Chapter 5 – Modules

5.1 The Big Picture

When your program gets bigger and you write more code, you will get tired of using the interpreter. This is because the code that you created in the interpreter is only valid in that session. Once you quit the interpreter you can never use that code again. Wouldn't it be nice if you can reuse code you wrote in different sessions?

With modules you can!

Besides that, modules are used to:

- Separate code with different concerns
- Organize code
- Reuse code made by someone else

5.2 Introduction

When you quit the Python interpreter, all the function definitions and variables you have created are lost, as you can see in the example below:

```
>>> message = "hello,world"
>>> def helloworld():
...     print message
...
>>> helloworld()
hello,world
>>>
>>> quit()
$ python
Python 2.6.4rc2 (r264rc2:75497, Oct 20 2009, 02:55:11)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> helloworld()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'helloworld' is not defined
```

```
>>>
```

To be able to use them, you would have to rewrite those function definitions and variables again. To save yourself some time, you can write your function definitions and variables in a file. By doing this you can reuse them in the different programs you write.

5.3 Defintion

In Python, **modules** are files that contain function definitions, variables and statements.

You can use your favorite text editor to write your modules. These files are saved with a .py extension. There are two ways to use python modules: either import them or execute them as a script.

5.4 Importing Modules

Copy the following code into a text editor and save as functions.py:

```
>>> def factorial(num):
...     factorial = 1
...     while num>0:
...         factorial *=num
...         num -=1
...     return factorial
>>> def square_sequence(n):
...     result=[]
...     for num in range(1,n+1):
...         result.append(num*num)
...     return result
... 
```

You can now make use of this function even after you quit the python interpreter but first,

- you have to make sure that in the terminal you are in the current directory where you saved your module
- you have to import the module into the current session

Okay, let us use the module we just saved in an interactive session:

```
>>> import functions

>>> functions.factorial(3)
```

You may assign the imported function to a local variable if you think "functions.factorial(...)" is too long:

```
>>> fact=functions.factorial
>>> fact(3)
```

But what if I need only the `square_sequence` function in the session and not the `factorial` function?

A variation of the `import` statement allows you only to import the `square_sequence` function:

```
>>> from functions import square_sequence
>>> square_sequence(3)
[1, 4, 9]
```

You can also import all of the functions, denoted by an asterisk, this way if you're feeling lazy:

```
>>> from functions import *
>>> factorial(3)
>>> square_sequence(3)
[1, 4, 9]
>>>
```

Note however that this practice is frowned upon because it makes the code less readable.

Other modules can also import modules. This is done, for example, when a module needs a function that is with another module.

To illustrate, consider the formula given in Chapter 2,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \text{ for all } x$$

The following is a function that approximates the sine of `x`:

```
from math import factorial, pi

def sin(x, t=10):
    sign_of_x = 1.0 if x >= 0 else -1.0
    x = float(abs(x)) % (2.0*pi)
    s = 0.0
    for i in range(1, 2*t, 2):
        sign = -1.0 if (i-1) % 4 else 1.0
        s += sign * x**i / factorial(i)
    return sign_of_x*s
```

Copy and paste the code above into a text editor and save it as `trig.py`.

Then, import your module and test it:

```
>>> from trig import sin
>>> sin(1)
0.8414709848078965
```

Notice that the function has a second argument, *t* which controls the number of terms in the expansion of the series. Experiment with different values of *t* and see what size of *t* will cause the function to return the same values as the function of the same name in the math library. Also, see if you can rewrite this function into a one-line lambda using a comprehension.

5.5 Executing Modules as scripts

Earlier, we said that modules can be run as a script. This means that we can call the function `sin(x)` in the module itself with a given parameter. We can do away with the importing it in the interpreter and entering the parameters ourselves. In fact, we are going to take it a step further, we are going to make our `trig` module ask input from the user!

To do this, we add the following code at the end of our file `trig.py`,

```
if __name__=="__main__":
    x = input('x?') #asks user for an input that gets assigned to x
    print "sin("+str(x)+")="+str(sin(x))
```

The conditional statement `if name==__main__`: lets our `trig.py` act as a standalone program. When you're using the interpreter think of it as the `__main__` at that moment.

Think of `__main__` as containing the code that we currently want to execute. The interpreter executes everything contained in `__main__`.

To execute `trig.py` as a module we feed it as a parameter when we fire up the interpreter,

```
oem@orange ~/Desktop/Sample Code $ python trig.py
x?2
sin(2)=0.909297426826
```

Make sure to change to the directory where `trig.py` is before running this command.

The difference is that in the case above, the `trig.py` module's name becomes `__main__` and not `trig`, however, when we import it in the interpreter to use `sin`, as you can see below:

```
>>> import trig
```

```
>>> trig.sin(2)
0.90929742682568171
>>> trig.__name__
'trig'
>>>
In fact, the __main__ at that moment is the interactive interpreter,
>>> __name__
'__main__'
```

5.6 The dir() function

The dir() function returns a list of strings that lists down all the names in a module from variables, functions and even imported modules.

Consider our trig module,

```
>>> import trig
>>> dir(trig)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'factorial',
'sin']
```

Expect each string to return something when you call them. We've already discussed the sin and factorial methods as well as the __name__ attribute.

The __doc__ attribute returns nothing because we didn't code in a doc string in the trig module:

```
>>> trig.__doc__
```

The __file__ attribute returns the name of the compiled version of the module:

```
>>> trig.__file__
'trig.pyc'
```

Without an argument, the dir() function returns a list of names --- variables, functions and modules --- that you have defined:

```
>>> numbers = range(1,5)
>>> import trig
>>> sin_func=trig.sin
>>> sin_func(1)
0.8414709848078965
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'numbers', 'sin_func',
'trig']
```

As you can see, the names we created are there: `numbers`, `sin_func` and `trig`. You can infer by now that the names `__builtins__`, `__doc__`, `__name__` and `__package__` are names included by default. Refer to the official Python documentation for more information about them.

5.7 Packages

Packages provide a way to organize your modules.

It's similar to the way that we organize our songs into genres in our phones or on our file system. There's a folder or play list for say, rock music and another one for classical and another one for alternative.

Creating packages are not all that hard. Packages are just different folders containing modules on your file system. For example,

```
chapter
  __init__.py
  trig/
    __init__.py
    sin.py
    cos.py
  factorial_script.py
  others/
    __init__.py
    fact.py
    square_sequence.py
```

The `__init__.py` inside each folder makes Python recognize the folder as a Python package. If a folder doesn't have an `__init__.py`, then Python simply sees it as a folder.

5.7.1 Importing modules from packages

Let's say I want to write a separate module that takes an input `x` and gets the factorial of `x`. We've already written the factorial function in `fact.py` before and so we can reuse that! We can import modules in the interpreter or from another module,

To do so using a module,

```
from chapter5.others.fact import factorial

if __name__ == '__main__':
    x = input('x?')
```

```

    print factorial(x)
# you can also do it this way: import chapter5.others.fact but you would have
to qualify
# the function as chapter5.others.fact.factorial(x) but that's way too long
and
# you wouldn't want that would you?

```

Make sure that the `factorial_script.py` is in same folder where your “chapter5” folder is placed:

```

Sample Code/
  chapter5/
    __init__.py
    ...
    ...
    others/
      factorial_script.py

```

Running the script,

```

oem@orange ~/Desktop/Sample Code $ python factorial_script.py
x?3
6

```

Using the interpreter,

```

>>> from chapter5.others.fact import factorial
>>> factorial(3)
6

```

or similarly,

```

>>> import chapter5.others.fact
>>> chapter5.others.fact.factorial(3)
6

```

5.7.2 Modules referencing from other modules

There is a problem with our existing `trig.py` code, where it attempts to refer to files in the same folder that are no longer there:

```

from functions import factorial
def sin(x):
    sequence=[1]
    for i in range (2,20):
        if i%2:
            sequence.append((-1.0 if sequence[-1]>0 else 1.0)
                            *x**i)

```



```

                                /fact(i))

    return sum(sequence)

if __name__ == "__main__":
    x = input('x?')
    print "sin("+str(x)+")="+str(sin(x))
sin.py

```

In section 5.4 our “sin.py” and “fact.py” were in the same folder,

```

chapter5/
    sin.py
    functions.py

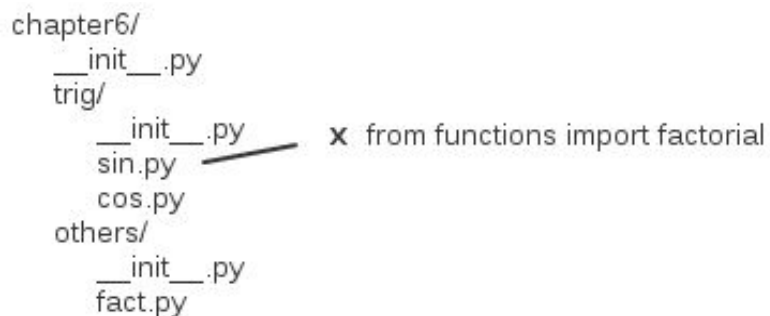
```

That is why there is no problem with using the directive “from function import factorial”. But we have reorganized our code into packages since then. We placed our sin module in the chapter5.trig package and the factorial function from functions.py to fact.py in the chapter5.others package.

```

chapter6/
  __init__.py
  trig/
    __init__.py
    sin.py
    cos.py
  others/
    __init__.py
    fact.py

```

 x from functions import factorial

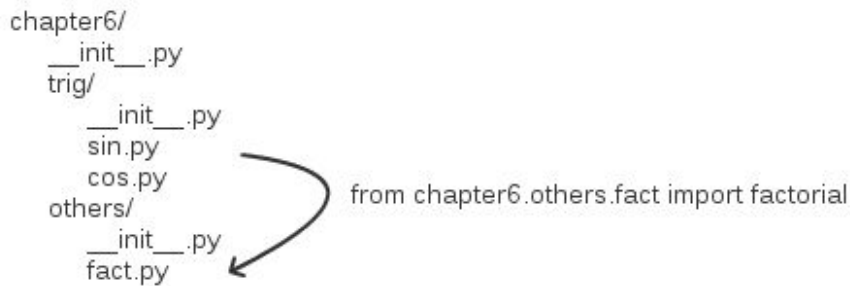
Thus, our current code won't work because it is trying to import the factorial function from the functions module but it is not there.

```

oem@orange ~/Desktop/Sample Code $ python
Python 2.6.4rc2 (r264rc2:75497, Oct 20 2009, 02:55:11)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from chapter5.trig.sin import sin
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    from functions import factorial
  File "chapter5/trig/sin.py", line 1, in <module>
    from functions import factorial
ImportError: No module named functions
>>>

```

To make it work, we need to import the factorial function from chapter5.others.fact:



This means that instead of starting with:

```
from functions import factorial
```

We change the first line to what you see below:

```
from chapter5.others.fact import factorial
def sin(x):
    sin_x=0
    for n in range(1,20):
        ...
    ...
```

Now, our program works again!

```
>>> from chapter5.trig.sin import sin
>>> sin(1)
0.8414709848078965
>>>
```

5.8 How does the interpreter go about looking for your files?

Whether you are coding from the interpreter or using modules, whenever you import a module, the Python interpreter first looks in the current directory for that module. If the module is not present, then it goes on to look at each directory in the list `sys.path`.

To see the list of directories in the `sys.path`:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2',
'/usr/lib/python2.6/lib-tk', '/usr/lib/python2.6/lib-old',
'/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/dist-packages',
'/usr/lib/python2.6/dist-packages/PIL',
'/usr/lib/python2.6/dist-packages/gst-0.10', '/usr/lib/pymodules/python2.6',
```

```
['/usr/lib/python2.6/dist-packages/gtk-2.0',  
 '/usr/lib/pymodules/python2.6/gtk-2.0',  
 '/usr/local/lib/python2.6/dist-packages']  
>>>
```

6

Chapter 6 – Object Oriented Programming

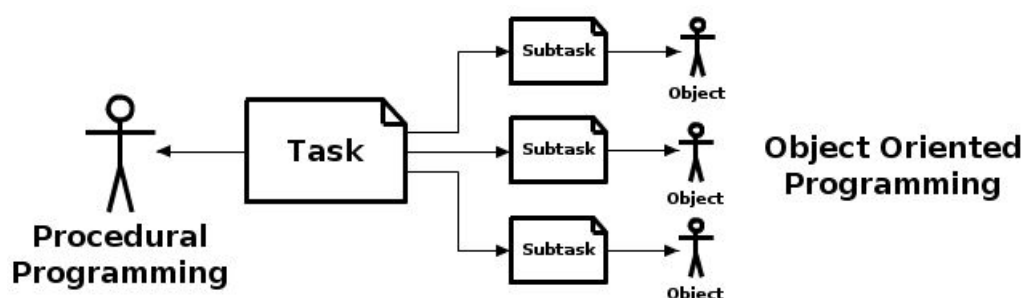
6.1 The Big Picture

In the earlier chapters, we programmed by making a function and then invoking it. Sometimes we would just write the code step by step without making a function. This form of programming is known as imperative or procedural. Imperative programming is good for small programs but as your program becomes larger and the specifications become more complex, a new approach is required. Thus, researchers developed Object Oriented Programming (OOP) to address that problem.

Object Oriented Programming is a **programming style** where an ecosystem of objects cooperate to perform a task.

6.2 Object Oriented Programming versus Procedural Programming

Think of procedural programming as one person that does a task. If the task becomes more complex, it is harder for just one person to execute that task. Adding someone to help him, the task becomes half as complex. If he can enlist more help then he can break down the task into sub tasks and delegate them. This is the idea behind object oriented programming.



Look at the things around you. You are probably sitting on a chair right now or lying down in bed. The chair and the bed are objects. This book you're reading is also an object! Look outside. There may be cars driving by or a dog running around. These too are objects. Things we see in the real life are objects. You can also be considered as an object!

What other real-life entities can you think of? These can probably all be represented in the computer as objects.

An **object** is usually a real life entity that the you, the programmer, tries to represent on the computer. It has attributes and behaviours.

Now that we have identified and defined what objects are, let us talk about attributes.

For example, a car's attributes are its make, model, color, seating capacity. Dogs belong to a certain breed. They come in different colors, Some have fur, and some have no fur at all. As a student you have a name and student number and a list of subjects that you are taking.

Attributes define an object. These are what make an object unique. They usually come in a set which is unique to that object. For example, a dog's attributes can be its breed and name. Although a student has a name as well, it doesn't have a breed.

Attributes are an object's characteristics.

Aside from attributes an object also has **behaviours**. Dogs can *sit*, *stand*, *rollover*, *bite* and *fetch* a frisbee. You, as a person, exhibit a variety of behaviours. You go to *sleep*, *eat*, *study*, *play*, etc. Right now you are *reading* this book. You might have some sort of beverage next to you that you can *drink*.

There are so many behaviours that an object can perform! In the next section we will make the connection between real life objects and objects in programs.

Behaviours are actions that an object can perform.

6.3 Classes

Before we can come up with objects in a program, we must specify how an object must look like. This can be done by coming up with a class.

A **class** is a blueprint of an object. It contains the the attributes and/or behaviours of an object.

A class is not the object itself but rather the design of the object.

Let's start making a student object! To code a class the most basic syntax for a class definition is:

```
class ClassName:
    def method_name(self, [optional arguments]):
        <statements>
```

Let's name our student class "Student":

```
>>> class Student:
...     pass
```

The code above follows the basic syntax in Figure 6.1 for creating a class.

The **pass** statement says that, right now, the Student class has nothing in it --- no attributes nor behaviours. We put the pass statement because the syntax above requires that a statement should be present. Since, at this point, we don't want to do anything, we put the pass statement which simply means that it will do nothing.

6.3 Creating an Object

Now we have a student class. Let us start by creating students.

```
>>> class Student:
...     pass
...
>>> s1=Student() #Creates s1 as an instance of Student
>>> s2=Student() #Creates s2 as an instance of Student
```

We now have two students! What we just did was to **instantiate** two instances of class Student.

Instantiation refers to creating a new object that follows the design and characteristics of a certain class.

In this case, we instantiated two objects of the Student class. `student_one` and `student_two` are both called instances of class Student.

6.4 Attributes

The two instances we created above are not very useful. They do not yet have names or student numbers and we don't know what subjects they're taking.

To make it look more like a student let us add some attributes:

```
>>> class Student:
...     pass
...
>>> s3=Student()
>>> s3.name="Jose"
>>> s3.student_number=1234
>>> s3.subjects=["Math","English"]
>>> print "s3 student's name:", s3.name
s3 student's name: Jose
>>> print "s3 student number:", s3.student_number
s3 student number: 1234
>>> print "s3 student's subjects:", s3.subjects
```

```
student's subjects: ['Math', 'English']
```

Initially, our Student class has no attributes. If we were to call `s3.name` before setting it, then that would cause an `AttributeError`:

```
>>> s3.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Student instance has no attribute 'name'
>>>
```

Alternatively, you might want to instantiate objects whose characteristics have already been customized in the class (`Student`). This can be done by defining a special method called `__init__(self, ...)`, and is often what you see in structured languages like Java. The attributes of a class are set first; they do not change, only their values can.

Whenever a class is created, the `__init__(self, ...)` method is called to set a newly created class to a specific state. The `__init__(self, ...)` method takes in a required parameter, **self** and other parameters that are needed to set the state of a new instance of the class. Consider the code below:

```
>>> class Student:
...     def __init__(self, name, student_number, subjects):
...         self.name = name
...         self.student_number = student_number
...         self.subjects = subjects
... 
```

The function `__init__(self, ...)` which takes in the required parameter `self`. Think of `self` as referring to the new class being created at that time. In fact, `self` is just a convention. You can use the word `me` or `this`, just like in Java, to refer to the instance. Note that when writing the `__init__()` method, `"__"` is a double underscore. In the example above, `__init__` is used to set a name, `student_number` and a list of subjects. This automatically gives instances of the class `Student` the attributes, `"name"`, `"student_number"` and `"subjects"`, whose values can be set at the time of instantiation.

For example, the code below creates a new instance of class `Student` with name `"Jose"`, student number `"1234"`, taking the subjects `"Calculus"`, and `"Programming"`.

```
>>> s1 = Student(name="Jose", student_number=1234, subjects=["Calculus",
"Programming"])
```

To make things easier, if you know the order of arguments when creating an instance, you do not have to write out each of the attributes' names:

```
>>> s2 = Student("Dan", 5678, ["Fine Arts", "Communication"])
```

Attributes are essentially “public”. There is no such thing as a truly “private” attribute in Python. However, if you insist, prefix any attribute with a double underscore. (There is always a way to get at these in Python, using introspection, although that is outside the scope of this introductory text.)

```
>>> class Student:
...     def __init__(self,name):
...         self.__name=name
...
```

For example, trying to access the private variable “__name” will result in an error:

```
>>> student_one = Student("Jose")
>>> student_one.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Student instance has no attribute '__name'
```

If you're coming from a statically-typed background, you might be wondering why there are no private variables in Python. The simple answer is that this is just a part of the Python's programming philosophy.

When an attribute is prefixed this way (i.e., with a double underscore “__”), it does not mean that you cannot access it at all. It simply means that the attribute is part of the implementation detail of say for instance, a method or a class. There are certainly ways to hack or to snoop around the privacy of a “private” variable and to access it. But this may make things too complicated, so, in the end, it is often better to just keep all of the attributes “public”.

Just in case you need to know how to access “private” attributes in Python, you will need to add a method that retrieves the name:

```
>>> class Student:
...     def __init__(self,name):
...         self.__name=name
...     def get_name(self):
...         return self.__name
...
>>> student_one = Student("Jose")
>>> print student_one.get_name()
'Jose'

>>> print student_one.name
'Jose'
```

Additionally, you can include a setter method if you want to allow for the changing of “private” attributes:

```
>>> class Student:
```



```

...
...
...     def set_name(self,name):
...         self.__name=name

>>> student_one = Student("Jose")
>>> print student_one.get_name()
Jose
>>> student_one.set_name("Dan")
>>> print student_one.get_name()
Dan

```

Simulating private attributes like this and adding methods to retrieve and set them is so common that Python has a shortcut, the `property()` function. Take a moment to look it up in the official documentation and rework the example to use it.

6.5 Behaviours

Now that we have made our student objects look more student-like, let's make them *behave* more like students. We begin by writing functions to reflect “Student” behaviours. One common way is to assign functions defined outside of the class as a local variable of that class.

```

>>> def study():
...     print "studying"
...
>>> a_student.study = study
>>> a_student.study()
studying

```

Functions can also be coded when you define a class:

```

>>> class Student:
...     def __init__(self,name,sid,subjects):
...         self.name=name
...         self.sid=sid
...         self.subjects=subjects
...     def study(self):
...         print "studying now"
...     def ask(self,question):
...         print question
...     def listen(self):
...         print "ZZZZZZZZ"

```

The following lines show our student instance executing its student methods. Now our student is acting like a student! Output is shown below.

```
>>> s1=Student("Jose",1234,["Calculus","Programming","Biology"])
>>> s1.listen()
ZZZZZZZZ
>>> s1.ask("What is the square root of 4?")
What is the square root of 4?
>>>
```

6.6 Inheritance

One of the main concepts in Object Oriented Programming is **inheritance**.

Inheritance is way to create a new class from an existing class.

6.6.1 Extending Created Classes

For example if we want to create a new class called `CollegeStudent`, we could code it from scratch like this:

```
>>> class CollegeStudent:
...     def __init__(self,name,sid,subjects):
...         self.name=name
...         self.sid=sid
...         self.subjects=subjects
...     def study(self):
...         print "studying now"
...     def ask(self,question):
...         print question
...     def listen(self):
...         print "ZZZZZZZZZ"
```

However, did you notice that `CollegeStudent` is almost the same as our `Student` class in Section 6.5? Instead of defining a completely new class for `CollegeStudent`, a much better way may be to “reuse” or “inherit” the code from the `Student` class.

The syntax for inheritance in Python goes like this:

```
class ClassName(ParentClassName):
    def method_name(self,[optional arguments]):
        <statements>
```

The “parent class” refers to the class whose code is to be inherited. In our case, the `CollegeStudent` class makes use of `Student` class to define itself. This makes `Student` the “parent class” as `CollegeStudent` the “subclass”.

Usually **subclasses are created because** we need to put in new attributes or behaviours that may or may not necessarily apply to the parent class. Think of the parent class as a more general type of a class. Subclasses are more specific.

Going back to our example,

```
>>> class CollegeStudent(Student):
...     def __init__(self, degree, major):
...         self.degree=degree
...         self.major=major
... 
```

The subclass `CollegeStudent` inherits the parent class `Student`. This means that:

- `CollegeStudent` has the attributes of `Student` -- `name`, `student_number` and `list_of_subjects`
- `CollegeStudent` exhibits the behaviours of `Student` -- `study`, `ask`, `listen`.
- `CollegeStudent` is given additional attributes, “degree” and “major” that the parent class, `Student` does not have.

Like adding new attributes specific to a college student, we can add new *behaviours* that are specific to a college student:

```
>>> class CollegeStudent(Student):
...     def __init__(self, degree, major):
...         self.degree=degree
...         self.major=major
...     def cut(self, class):
...         print "feeling lazy, won't go to %s class" % class
...     def study():
...         print "sipping coffee while going through a ton of readings"
```

As you can see in the example above, the `CollegeStudent` was given the behaviours of “cut” and “study”, that are specific to `CollegeStudent` and not found in its parent class, `Student`.

Finally, subclasses can change behaviours that are inherited. This is called overriding a function. For instance we might want to change how the `study()` function is to reflect how a college student really studies.

```
>>> class CollegeStudent(Student):
...     def __init__(self, degree, major):
...         self.degree=degree
...         self.major=major
...
...     def cut(self, class):
...         print "feeling lazy, won't go to %s class" % class
...     def study(self):
...         print "studying while sipping coffee"
```

```
...
>>> college_student = CollegeStudent("Jose",1234,["Calculus","Computer
Programming",])
>>> college_student.study()
studying while sipping coffee
>>>
```

6.7 Exercises

1. Create a Section class with a section id, course title, instructor and students as class variables.
2. Create an Instructor class that has a name and list of subjects being taught as its class variables.
3. All objects in Python inherit from the Object class. Override that class' `__contains__` method in the Section class such that we can do this:

```
>>> section=Section(1,"Algebra",Instructor("Raul"),["Jose","Luis"])
>>> "Jose" in section
True
>>>
```

4. Modify the Section class to allow students to be added to a section even after object creation. Also allow for checking of the current size of the section. For example,

```
>>> section=Section(1,"Algebra",instructor,["Jose","Luis"])
>>> section.add("Roberto")
>>> section.size()
3
```

5. Allow iterating over a section's students such as:

```
>>> for student in section:
...     print student
Jose
Luis
Roberto
```

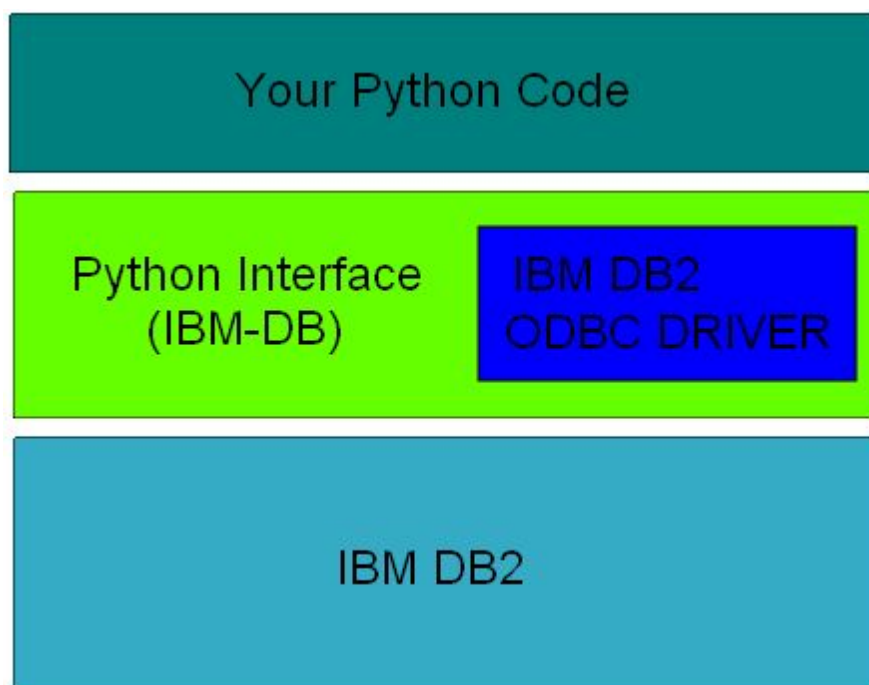
7

Chapter 7 – Database Connectivity

7.1 The Big Picture

The Python DB API provides a common interface to access relational databases like DB2. With this specification, our Python programs can access different databases using the same syntax without knowing details of each specific database. It is up to the drivers of each database to handle these details for us. There are a lot of DB2 drivers for Python. In this book we will use the driver provided by IBM.

The following figure illustrates how our Python programs interact with relational databases.



Our Python code contains our SQL queries. To be able to access IBM's DB2, it asks help from IBM DB -- an implementation of the Python DB API 2.0. IBM DB contains the code that will take our queries and run it on IBM DB2. To do this, it uses the the IBM DB2 ODBC Driver that is wrapped with it.

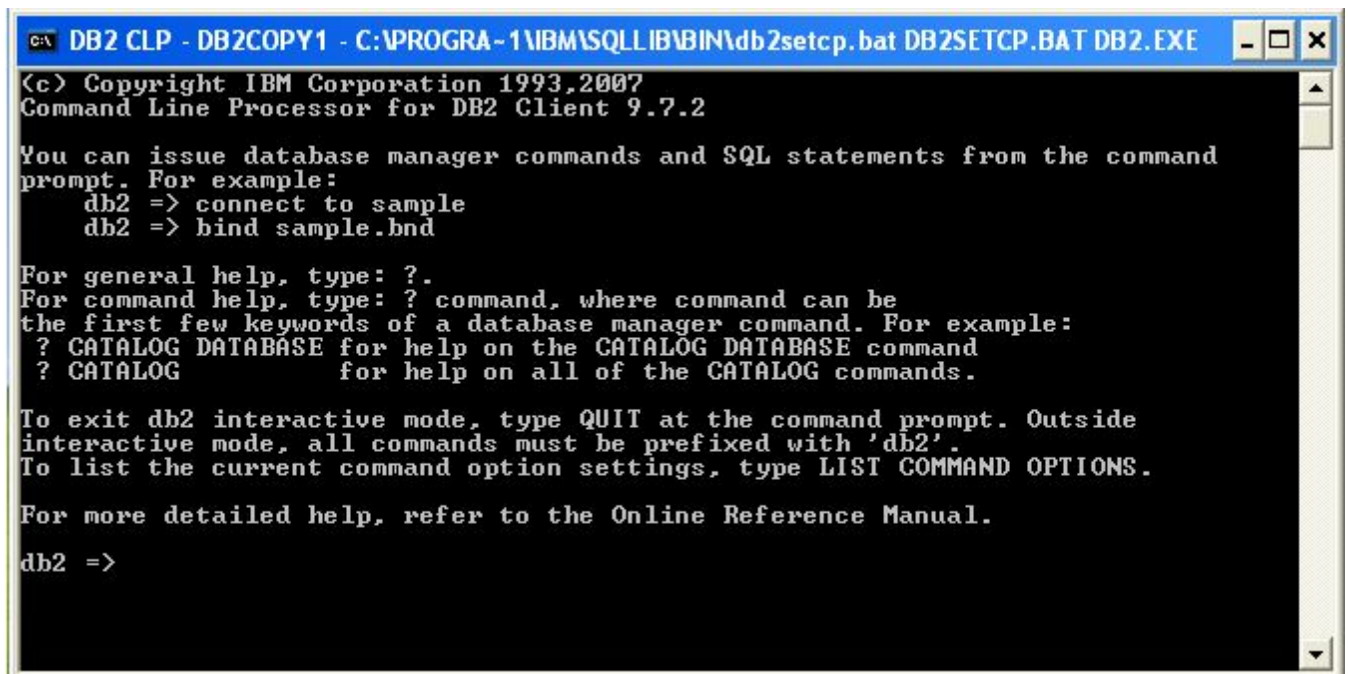
7.2 Prerequisites

Before we can use DB2, there are several things that we have to do:

- Get a copy of IBM DB2 Express-C.. There are many ways to get DB2 but the easiest is from the website at [DB2 Express-C Data Server](#)
- Install IBM DB2 Express-C.
- Install easy_install, which is found here: [Python Setup Tools](#)
- Using the latter, install the IBM-DB egg, an implementation of the Python DB API 2.0 so that we can interface with DB2 using Python. You can download it from <http://code.google.com/p/ibm-db/> (Take note: Do not download the IBM-DB egg for Django but rather the IBM-DB egg. In our case since we're using Python 2.6, the filename looks something like ibm_db-1.0.3-py2.6-win32.egg)

7.3 Setting up

Go to the command processor. You should see this screen once you are there (your DB2 version may be different):

A screenshot of a Windows command prompt window titled "DB2 CLP - DB2COPY1 - C:\PROGRA~1\IBM\SQL\IBMDBIN\db2setcp.bat DB2SETCP.BAT DB2.EXE". The window shows the DB2 Command Line Processor (CLP) interface. It displays copyright information for IBM Corporation (1993, 2007) and identifies the version as DB2 Client 9.7.2. It provides instructions on how to use the command prompt, including examples of database manager commands like 'connect to sample' and 'bind sample.bnd'. It also lists keywords for help, such as 'CATALOG DATABASE' and 'CATALOG'. Finally, it shows the prompt 'db2 =>' ready for user input.

```
C:\ DB2 CLP - DB2COPY1 - C:\PROGRA~1\IBM\SQL\IBMDBIN\db2setcp.bat DB2SETCP.BAT DB2.EXE
(c) Copyright IBM Corporation 1993,2007
Command Line Processor for DB2 Client 9.7.2

You can issue database manager commands and SQL statements from the command
prompt. For example:
    db2 => connect to sample
    db2 => bind sample.bnd

For general help, type: ?.
For command help, type: ? command, where command can be
the first few keywords of a database manager command. For example:
? CATALOG DATABASE for help on the CATALOG DATABASE command
? CATALOG           for help on all of the CATALOG commands.

To exit db2 interactive mode, type QUIT at the command prompt. Outside
interactive mode, all commands must be prefixed with 'db2'.
To list the current command option settings, type LIST COMMAND OPTIONS.

For more detailed help, refer to the Online Reference Manual.

db2 =>
```

Next, let's create a database:

```
db2 => create database sample
DB20000I  The CREATE DATABASE command completed successfully.
```

Connect to the database we just made.

```
db2 => connect to sample
```

Database Connection Information

Database server = DB2/NT 9.7.2

SQL authorization ID = JOSE

Local database alias = SAMPLE

Let's now create a table:

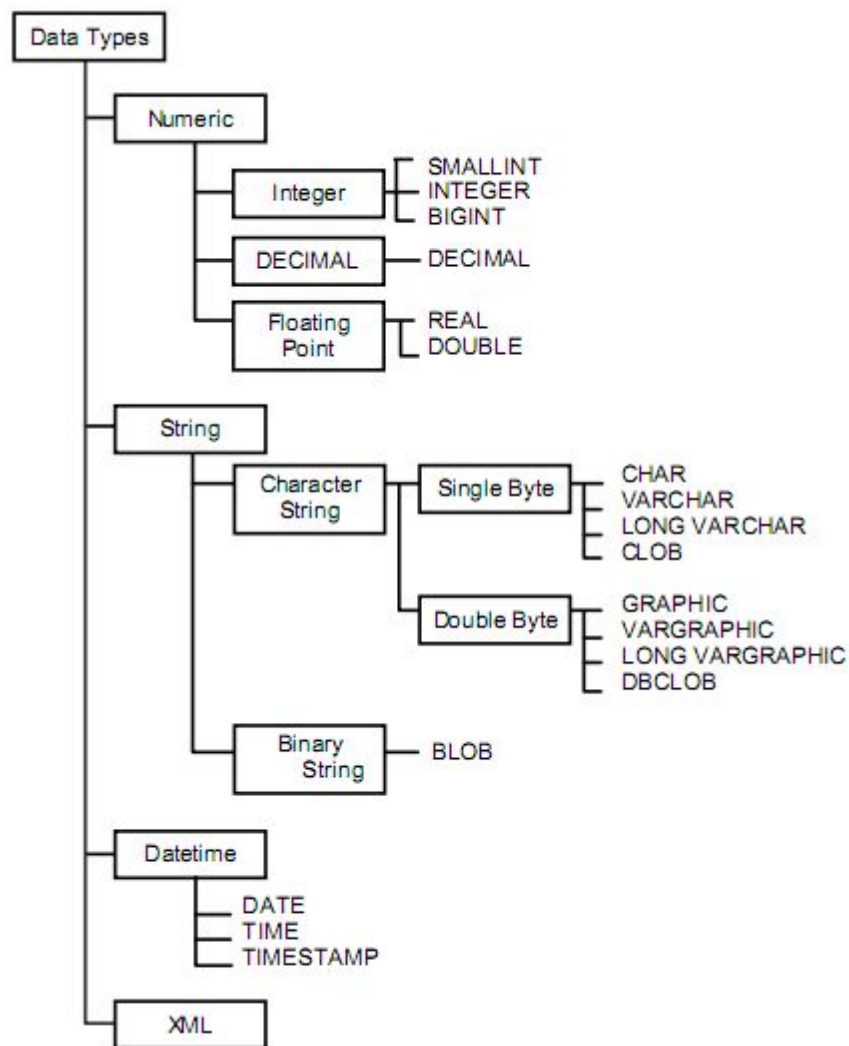
```
db2 => create table student(id smallint not null,
                           last_name varchar(100),
                           given_name varchar(100),
                           login_name varchar(100),
                           year_level smallint,
                           birthday date,
                           home_address varchar(100))
```

DB20000I The SQL command completed successfully.

Note: Any table you create will have your userid added to the front as a schema name. For example, if you were logged in using the username Zaphod, your student table would have the full name "Zaphod.student" Alternatively, you can specify a schema name when you create your table, e.g.:

```
create table dent.student
```

Some other data types in IBM DB2 are shown in the picture below:



We need just one more step to complete our setup. Let's populate our database with rows:

```
db2 =>INSERT INTO student VALUES (1, 'Asuncion', 'Jose',  
'jasuncion',2,'1988-04-08','Manila')
```

```
DB20000I The SQL command completed successfully.
```

```
db2 =>INSERT INTO student VALUES (1, 'Delgado', 'Dan',  
'dcdelgado',4,'1988-04-20','Manila')
```

```
DB20000I The SQL command completed successfully.
```

If we wanted to populate our database with many values at a time, all we have to do is:

```
db2 => INSERT INTO student VALUES (3, 'Duffy','Arthur','aduff',4,
'1989-08-15', 'Bulacan'), (4, 'Watson', 'Emilie', 'eawatson', 5, '1987-3-15',
'Singapore')
DB20000I The SQL command completed successfully.
```

7.4 Using IBM DB2 with Python

Now that we are all set up, it's time to hit the database! To be able to use IBM DB2 with Python we have to follow some simple steps:

- Import the DB2 module and connect to the database
- Get a connection object
- Create a cursor
- Perform SQL queries
- Close the cursor and the connection

7.5 Connection Objects

The first step in accessing a DB2 database is to import the IBM DB2 Python module that we have installed.

```
>>> import ibm_db
```

Then, we establish a connection to a particular data source.

```
>>> ibm_db_conn=ibm_db.connect("sample","your name","your password")
```

The connect() constructor accepts the following parameters:

- dsn - data source name, required - in our example our datasource is the database we created
- uid - user name, required
- pwd - password, required

This is similar to when we connected to the database in the command line processor in an earlier section:

After that let us retrieve a connection object:

```
>>> import ibm_db_dbi
>>> conn = ibm_db_dbi.Connection(ibm_db_conn)
>>>
```

7.6 Cursor Objects

Once you have a connection to the database, you can use it to retrieve a cursor.

```
>>> cursor = conn.cursor()
```

A cursor makes us work with the IBM DB2 database. For example, it has methods which will take in our SQL STATEMENTS (SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER) as a parameter and send them to the database. We will discuss these in more detail in the next section.

7.7 Performing SQL queries

Before we start, let us first add this method definition to the namespace.

```
>>> def printrows(rows):
    for row in rows:
        out = ""
        for data in row:
            out+=str(data)+' '
        print out
```

7.7.1 SELECT

The first thing that comes to mind regarding SQL is the SELECT statement. Let us get started with it in Python! Use the same session that we used in the setup part. If you have already closed that session, open a new one and then make a new connection.

```
>>> cursor.execute("select * from student")
>>> rows=cursor.fetchall()
```

The cursor's `execute()` method takes in our `SELECT` statement and sends it to the database. The `fetchall()` method gets all the possible rows that matches our query. Finally, we can output to the screen the results of our query:

```
>>> printrows(rows)

1  Asuncion   Jose    jasuncion   2    1988-04-08   Manila
2  Delgado   Dan     dcdelgado   4    1988-04-20   Manila
3  Lacanilao Arturo  aclacanilao  4    1989-08-15   Bulacan
4  Watson    Emilie  eawatson    5    1987-03-15   Singapore
>>>
```

You can use the `fetchone()` method to get the rows one by one:

```
>>> while True:
...     row = curs.fetchone()
...     if row is None:
...         break
...     print row
```

What this does is to start from the first row of the result set and then give the option to iterate over the whole result set one by one --- which is essentially stored in a list.

If you want, a parameter can specify the number of rows you want to retrieve by adding a numerical parameter to the `fetchone()` method.

```
>>> rows = cursor.fetchmany(3)
>>> printrows(rows)

1  Asuncion Jose jasuncion 2 1988-04-08 Manila
2  Delgado Dan dcdelgado 4 1988-04-20 Manila
3  Lacanilao Arturo aduff 4 1989-08-15 Bulacan
```

Maybe you would like to add a filter to the query. Just construct a new query as a string.

```
>>> query2 = "select * from artists where last_name='Asuncion'"
>>> cursor.execute(query2)
>>> rows = cursor.fetchall()
>>> print(rows)
1  Asuncion   Jose    jasuncion   2    1988-04-08   Manila
>>>
```

7.7.2 INSERT

The INSERT statement is used to add more rows into a table.

Naively, the following method works:

```
>>> cursor.execute("INSERT INTO student VALUES  
                    (1, 'Webb', 'Ryan', 'rgwebb', 5, '10-7-1984', 'London')")
```

Let us take that a step further and save ourselves some typing space so we don't have to key in the whole "cursor.execute(...)" line of code whenever we want to do an insert. By using dictionaries we can pass a whole row as a parameter to our Python insert statement:

```
>>> new_entry=(6, "Man", "Spider", "scparker", 5, "1990-1-1", "New York")  
>>> cursor.execute("insert into student values (?, ?, ?, ?, ?, ?, ?)", new_entry)
```

Alternatively, we can insert multiple rows in a single execute() call. However, instead of using the execute() method, we will use the executemany() method. Let's do that:

We make up a list of sample data to insert.

```
>>> list = [(i, 'Lastname', 'Firstname', 'lgnname', i, '1-1-2010', 'Hometown')  
            for i in range (10, 15)]
```

We form the query and the feed it as well as the list to the cursor's executemany() method:

```
>>> insert_sql = "INSERT INTO student VALUES (?, ?, ?, ?, ?, ?, ?)"  
>>> cursor.executemany(insert_sql, list)
```

Check to see if all the values were inserted by running a SELECT * query from the database.

7.8 Exercises

1. Create a program that allows a user to

- Create Section
- Add a list of students and save to the database
- View Sections
- View a list of students in a section
- Add a student/s to a section
- Remove a student from a section and update the database

2. Using the `format()` function discussed in chapter 3 and the `namedtuple` function discussed in chapter 4, write a Python program that will access the student table created in the example and produce a tabular report with column headings and all data left aligned.



Appendix A – Solutions to the review questions

Chapter 2

1.
 - a. `y = 2 * 3`
 - b.

```
if 4 == 2/2: # Check if 4 is equal to 2/2
    print "4 = 2/2!"
```
 - c.

```
for x in 1, 2, 3:    # print the first three positive integers
    print x
```
 - d.

```
x = 4 if b else x    # assign 4 to x if b is True
-- or --
if b: x = 4
```
 - e.

```
while True:
    x = 1
    if x: continue  # indentation must be the same
```
2.

```
n = 5
a, b = 0, 1
for i in range(n):
    a, b = b, a + b
print a
```
3.
$$x - x^{**3}/(3*2*1) + x^{**5}/(5*4*3*2*1) - x^{**7}/(7*6*5*4*3*2*1) + x^{**9}/(9*8*7*6*5*4*3*2*1) - x^{**11}/(11*10*9*8*7*6*5*4*3*2*1)$$
4.

```
m=6
months = (
    ('January'   , 'Garnet')
    , ('February' , 'Amethyst')
    , ('March'    , 'Aquamarine')
    , ('April'    , 'Diamond')
    , ('May'      , 'Emerald')
    , ('June'     , 'Alexandrite')
    , ('July'     , 'Ruby')
    , ('August'   , 'Peridot')
```

```

        , ('September', 'Sapphire')
        , ('October' , 'Tourmaline')
        , ('November' , 'Topaz')
        , ('December' , 'Zircon')
    )
    print 'Month {0} is {1} and it''s birthstone is {2}' \
        .format(m, months[m-1][0], months[m-1][1])

```

Chapter 3

1.

```

def fib_binet(n):
    if n == 0 : return 0
    phi = (1 + math.sqrt(5))/2
    F = int(math.floor(phi**n/math.sqrt(5) + .5))
    return F

```

2.

```

def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        cr.next()
        return cr
    return start

```

3.

```

def pywalk(dir):
    """
    os.Walk(top) For each directory in the tree rooted at directory
    top it yields a 3-tuple
    (dirpath, dirnames, filenames).
    """

    for (_, _, filenames) in os.walk(dir):
        for filename in filenames:
            if filename.endswith('.py'):
                yield filename

# Example usage
for filename in pywalk(sys.path[7]): print filename

```


4. (a)

```
from bisect import bisect_left
def listops(l, item, operation):
    if not isinstance(l, list):
        raise TypeError('First argument is not a list')
    if operation == 0:
        l.sort()
    elif operation == 1:
        i = bisect_left(l, item)
        l.insert(i, item)
        return i == len(l) or l[i+1] == item
    elif operation == 2:
        i = bisect_left(l, item)
        if i < len(l) and l[i] == item:
            del l[i:i+1]
            return True
        else:
            return False
    elif operation == 4:
        i = bisect_left(l, item)
        if i < len(l) and l[i] == item:
            return True
        else:
            return False
```

4. (b)

```
def c0(l, item): return listops(l, item, 0)
def c1(l, item): return listops(l, item, 1)
def c2(l, item): return listops(l, item, 2)
def c4(l, item): return listops(l, item, 4)
```

4. (c)

```
from functools import partial
mylistops = partial(listops, mylist)
```

Chapter 4

1.

```
import sys
import os
```

```

from collections import defaultdict

mydict = defaultdict(list)
for path in sys.path:
    if path == '': continue          # ignore empty paths

    # handle Windows drives
    if path[0].isalpha() and path[1] == ':':
        drive, path = path.split(':')
        drive += ':'
    else:
        drive = ''

    # assemble prefix for directory walk
    prefix = drive + os.sep.join(path.split(os.sep)[:3])

    # Directory walk
    for dirpath, dirnames, _ in os.walk(prefix):

        # Process subdirectories returned
        for suffix in dirnames:
            # assemble full path
            subdir = (os.sep.join([dirpath, suffix]))

            # remove prefix and add to dict
            subdir = subdir[len(prefix)+1:]
            mydict[prefix].append(subdir)

# print the results
for prefix in sorted(mydict):
    print prefix
    for suffix in sorted(mydict[prefix]):
        print '    ' + suffix

```

2. (a)

```

def movies(md):
    # Get movie title and director name
    while True:
        movie_name = raw_input('Movie name and year :')
        if movie_name == '': break
        director = raw_input('Director name: ')

```

```

    if director == '': break

    # Get cast members
    cast = []
    while True:
        cast_member = raw_input('Cast member: ')
        if cast_member == '': break
        cast.append(cast_member)
    md[movie_name] = (director, tuple(cast))

```

2. (b)

```

def moviestar(md, star):
    # Build list of movies star has been in
    movie_list = []
    for movie, (director, cast) in md.iteritems():
        if any(star == cast_member.split(' ')[-1]
               for cast_member in cast):
            movie_list.append(movie)

    # Print the list of movies
    for movie in movie_list:
        print 'Title:      ' + movie
        print 'Director: ' + md[movie][0]
        print 'Cast:      ' + md[movie][1][0]
        for cast_member in md[movie][1][1:]:
            print '          ' + cast_member
        print

```

3. (a)

```

from random import random
def buildarray(rows, cols, min, max):
    array = \
        [
            [min + random()*(max-min) for _ in range(cols)]
            for _ in range(rows)
        ]
    return array

```

3. (b)

```

def matrix_mult(A, B):
    if not all(len(col) == len(A[0]) for col in A[1:]):

```

```

        raise ValueError, 'Array A is not rectangular'
    if not all(len(col) == len(B[0]) for col in B[1:]):
        raise ValueError, 'Array B is not rectangular'
    if len(A) <> len(B[0]):
        raise ValueError, 'Arrays cannot be multiplied'

    AB = [[0 for _ in xrange(len(A))] for _ in xrange(len(A))]
        for i,j in ((i,j) for i in xrange(len(AB))
for j in xrange(len(AB))):
            AB[i][j] = sum(A[i][k]*B[k][i]
                            for k in xrange(len(B)))

    return AB

```

Chapter 6

1.

```

class Section:
    id = 0
    title = ''
    instructor = ''
    students = []

    def __init__(self, id, title, instructor, students):
        self.id = id
        self.title = title
        self.instructor = instructor
        self.students = students

```

2.

```

class Instructor:
    name = ''
    subjects = []

    def __init__(self, name, subjects=[]):
        self.name = name
        self.subjects = subjects

```

3. Add class method:

```

    def __contains__(self, name):
        return name in self.students

```

4. Add class methods:

```
def add(self, name):
    self.students.append(name)

def size(self):
    return len(self.students)
```

5. Add class method:

```
def __iter__(self):
    for student in self.students:
        yield student
```

Chapter 7

1.

```
# import needed modules
import ibm_db
import ibm_db_dbi

# connect to database and get cursor object
cibm = ibm_db.connect('sample', 'youruserid', 'yourpassword')
conn = ibm_db_dbi.Connection(cibm)
curs = conn.cursor()

# Define CREATE command for Section table
create_section = """
CREATE TABLE Section(
    Id INTEGER GENERATED ALWAYS AS IDENTITY
    , title varchar(50)
    , instructor varchar(50)
    , student varchar(50))
;
"""

# Create Section table
curs.execute(create_section)

# Define INSERT INTO command for Section Table
insert_section = """
INSERT INTO Section(title, instructor, student) VALUES
```

```

(?,?,?)
"""

# Insert students into Section table
title = 'Getting started with Python'
instructor = 'Gerald and Jose'
students = ('Bob', 'Carol', 'Ted', 'Alice')

for student in students:
    curs.execute(insert_section, (title, instructor, student))

# View sections

curs.execute('SELECT * FROM Section')
for row in curs.fetchall():
    print row

# View students in a section
curs.execute(
    'SELECT student FROM Section WHERE title = ?',
    [title])
for student in curs.fetchall():
    print student

# Add students to a section
new_students = ('Frodo', 'Bilbo', 'Gandalf', 'Galadriel')
for student in new_students:
    curs.execute(insert_section, (title, instructor, student))

# Remove student from a section
curs.execute(
    'DELETE FROM Section WHERE title = ? AND Student = ?',
    [title, 'Bob'])

```

2.

```

# Formatted output
from collections import namedtuple
StudentRow = namedtuple('StudentRow', 'ID, LAST_NAME,
GIVEN_NAME, LOGIN_NAME, YEAR_LEVEL, BIRTHDAY, HOME_ADDRESS')

curs.execute('SELECT * FROM STUDENT')

```

```

row_format = '{0:>2} {1:<12} {2:<12} {3:<12} {4:>4} {5:>10}
{6:<12}'
print row_format.format(
    'ID', 'LAST_NAME', 'GIVEN_NAME', 'LOGIN_NAME', 'YEAR',
    'BIRTHDAY', 'HOME_ADDRESS'
)

row_format = '{0:>2} {1:<12} {2:<12} {3:<12} {4:>4} {5!s:>10}
{6:<12}'

for stu in map(StudentRow._make, curs.fetchall()):
    print row_format.format(
        stu.ID, stu.LAST_NAME, stu.GIVEN_NAME,
        stu.LOGIN_NAME, stu.YEAR_LEVEL, stu.BIRTHDAY,
        stu.HOME_ADDRESS
    )

conn.close()

```

References

- [1] ZIKOPOULOS, P. *IBM® DB2® Universal Database™ and the Microsoft® Excel Application Developer... for Beginners*, dbazine.com article, April 2005
<http://www.dbazine.com/db2/db2-disarticles/zikopoulos15>
- [2] ZIKOPOULOS, P. *DB2 9 and Microsoft Access 2007 Part 1: Getting the Data...*, Database Journal article, May 2008 <http://www.databasejournal.com/features/db2/article.php/3741221>
- [3] BHOGAL, K. *Use Microsoft Access to interact with your DB2 data*, developerWorks article, May 2006.
<http://www.ibm.com/developerworks/db2/library/techarticle/dm-0605bhogal/>
- [4] CHUN, J., CIRONE P. *DB2 packages: Concepts, examples, and common problems*, developerWorks article, June 2006 <http://www.ibm.com/developerworks/data/library/techarticle/dm-0606chun/index.html>
- [5] CHEN Whei-Jen et all. *DB2 Express-C: The Developer Handbook for XML, PHP, C/C++, Java, and .NET* August 2006 - SG24-7301-00 <http://www.redbooks.ibm.com/abstracts/sg247301.html?Open>

Resources

Web sites

1. DB2 Express-C web site:
www.ibm.com/db2/express
Use this web site to download the image for DB2 Express-C servers, DB2 clients, DB2 drivers, manuals, access to the team blog, mailing list sign up, etc.
2. DB2 Express-C forum: www.ibm.com/developerworks/forums/dw_forum.jsp?forum=805&cat=19
Use the forum to post technical questions when you cannot find the answers in the manuals yourself.
3. DB2 Information Center
<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp>
The information center provides access to the online manuals. It is the most up to date source of information.
4. developerWorks
<http://www-128.ibm.com/developerworks/db2>
This Web site is an excellent resource for developers and DBAs providing access to current articles, tutorials, etc. for free.
5. alphaWorks®
<http://www.alphaworks.ibm.com/>
This Web site provides direct access to IBM's emerging technology. It is a place where one can find the latest technologies from IBM Research.
6. planetDB2
www.planetDB2.com
This is a blog aggregator from many contributors who blog about DB2.

7. DB2 Technical Support

If you purchased the 12 months subscription license of DB2 Express-C, you can download fixpacks from this Web site.

http://www.ibm.com/software/data/db2/support/db2_9/

8. ChannelDB2

ChannelDB2 is a social network for the DB2 community. It features content such as DB2 related videos, demos, podcasts, blogs, discussions, resources, etc. for Linux, UNIX, Windows, z/OS, and i5/OS.

<http://www.ChannelDB2.com/>

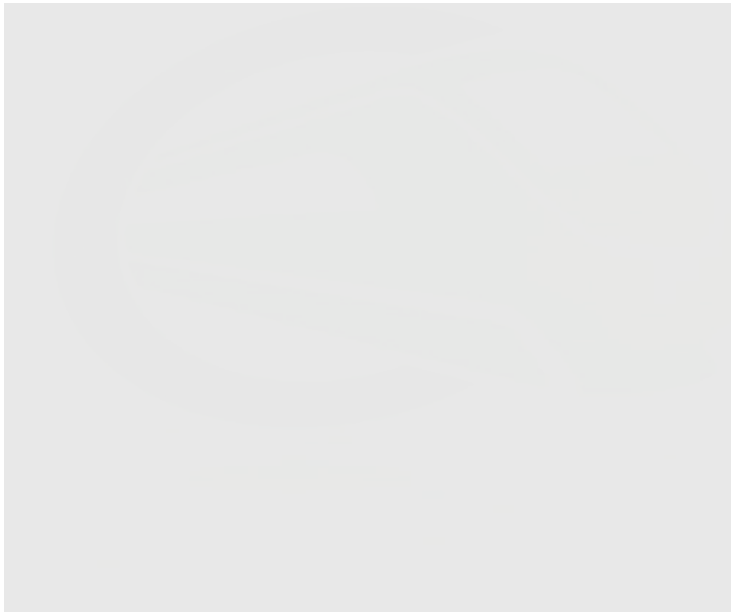
Books

1. Free Redbook: DB2 Express-C: The Developer Handbook for XML, PHP, C/C++, Java, and .NET
Whei-Jen Chen, John Chun, Naomi Ngan, Rakesh Ranjan, Manoj K. Sardana,
August 2006 - SG24-7301-00
<http://www.redbooks.ibm.com/abstracts/sg247301.html?Open>
2. Understanding DB2 – Learning Visually with Examples V9.5
Raul F. Chong, et al. January 2008
ISBN-10: 0131580183
3. DB2 9: pureXML overview and fast start by Cynthia M. Saracco, Don Chamberlin, Rav Ahuja June 2006 SG24-7298
<http://www.redbooks.ibm.com/abstracts/sg247298.html?Open>
4. DB2® SQL PL: Essential Guide for DB2® UDB on Linux™, UNIX®, Windows™, i5/OS™, and z/OS®, 2nd Edition
Zamil Janmohamed, Clara Liu, Drew Bradstock, Raul Chong, Michael Gao, Fraser McArthur, Paul Yip
ISBN: 0-13-100772-6
5. Free Redbook: DB2 pureXML Guide
Whei-Jen Chen, Art Sammartino, Dobromir Goutev, Felicity Hendricks, Ippei Komi, Ming-Pang Wei, Rav Ahuja, Matthias Nicola. August 2007
<http://www.redbooks.ibm.com/abstracts/sg247315.html?Open>
6. Information on Demand - Introduction to DB2 9 New Features
Paul Zikopoulos, George Baklarz, Chris Eaton, Leon Katsnelson
ISBN-10: 0071487832
ISBN-13: 978-0071487832

Contact emails

General DB2 Express-C mailbox: db2x@ca.ibm.com

General DB2 on Campus program mailbox: db2univ@ca.ibm.com



Getting started with DB2 application development couldn't be easier.

Read this book to:

- **Discover DB2[®] application development using DB2 Express-C**
- **Write SQL, XQuery, and understand pureXML[®] technology**
- **Learn how to develop DB2 stored procedures, functions and data Web services**
- **Learn how to work with DB2 and Java[™], C/C++, .NET, PHP, Ruby on Rails, Perl, and Python**
- **Troubleshoot DB2 database-related problems**
- **Practice with hands-on exercises**

DB2 Express-C from IBM is the no-charge edition of DB2 data server for managing relational and XML data with ease. No-charge means DB2 Express-C is free to download, free to develop your applications, free to deploy into production, and even free to embed and distribute with your solution. And, DB2 does not place any artificial limits on the size of databases, number of databases, or number of users.

DB2 Express-C runs on Windows[®], Linux[®], Solaris, and Mac OS X systems, and provides application drivers for a variety of programming languages and frameworks including C/C++, Java, .NET, Ruby on Rails, PHP, Perl, and Python. If you require even greater scalability or more advanced functionality, you can seamlessly deploy applications built using DB2 Express-C to other DB2 editions such as DB2 Workgroup and DB2 Enterprise.

This free edition of DB2 is ideal for developers, consultants, ISVs, DBAs, students, or anyone who intends to develop, test, deploy, or distribute database applications. Join the growing DB2 Express-C user community today and take DB2 Express-C for a test drive. Start discovering how you can create next generation applications and deliver innovative solutions.

To learn more or download DB2 Express-C, visit ibm.com/db2/express

To socialize and watch related videos, visit channelDB2.com

This book is part of the DB2 on Campus book series, free eBooks for the community. Learn more at db2university.com

Price: 24.99USD