

PYTHON

- 1. Features of Python**
- 2. Syntax and Style**
- 3. Python Objects**
- 4. Numbers**
- 5. Sequences**
- 6. Strings**
- 7. Lists**
- 8. Tuples**
- 9. Dictionaries**
- 10. Conditionals and Loops**
- 11. Files**
- 12. Input and Output**
- 13. Errors and Exceptions**
- 14. Functions**
- 15. Modules**
- 16. Classes and OOP**
- 17. Execution Environment.**

Features of Python

- 1. Simple**
- 2. Easy to Learn**
- 3. Free and Open Source**
- 4. High Level Language**
- 5. Portable**
- 6. Interpreted**
- 7. Object oriented**
- 8. Extensible**
- 9. Embeddable**
- 10. Extensive Libraries**

Simple

Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.

Easy to Learn

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

Free and Open Source

Python is an example of a FLOSS (Free/LibrÃ© and Open Source Software). In simple terms, you can freely distribute copies of this software, read it's source code, make changes to it, use pieces of it in new free programs, and that you know you can do these things. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

High-level Language

When you write programs in Python, you never need to bother ⁴ about the low-level details such as managing the memory used by your program, etc.

Portable

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

You can use Python on Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and even PocketPC !

Interpreted

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

Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc, etc. This also makes your Python programs much more portable, since you can just copy⁵ your Python program onto another computer and it just works!

Object Oriented

Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

Extensible

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

Embeddable

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

Extensive Libraries

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

INTRODUCTION

1. Python is a rich and powerful language, but also one that is easy to learn.
2. You can start the Python interpreter from the command line. Change to the directory where the interpreter lives, or add the directory to your path. Then type: `python` On UNIX, Python typically lives in `/usr/local/bin`; on Windows, Python probably lives in `c:\python20`.
3. Python creates variables the first time you use them (you never need to explicitly declare them beforehand), and automatically cleans up the data they reference when they are no longer needed.

SYNTAX AND STYLE:

This includes the following contents

1. Statements and syntax
2. Variable assignment
3. Identifiers and keywords
4. Basic style guidelines
5. Memory management
6. First Python application

Statements and Syntax

Some rules and certain symbols are used with regard to statements in Python:

1. Hash mark (#) indicates Python comments
2. NEWLINE (\n) is the standard line separator (one statement per line)
3. Backslash (\) continues a line
4. Semicolon (;) joins two statements on a line
5. Colon (:) separates a header line from its suite
6. Statements (code blocks) grouped as suites
7. Suites delimited via indentation
8. Python files organized as "modules"

Comments (#)

Python comment statements begin with the pound sign or hash symbol (#). A comment can begin anywhere on a line. All characters following the # to the end of the line are ignored by the interpreter. Use them wisely and judiciously.

Continuation (\)

Python statements are, in general, delimited by NEWLINEs, meaning one statement per line. Single statements can be broken up into multiple lines by use of the backslash. The backslash symbol (\) can be placed before a NEWLINE to continue the current statement onto the next line.

Example

```
# check conditions
if (weather_is_hot == 1) and \
(shark_warnings == 0):
    send_goto_beach_mesg_to_pager()
```

There are two exceptions where lines can be continued without backslashes. A single statement can take up more than one line when (1) container objects are broken up between elements over multiple lines, and when (2) NEWLINEs are contained in strings enclosed in triple quotes.

```
# display a string with triple quotes
print "hi there, this is a long message for you that goes over
multiple lines... you will find out soon that triple quotes in
Python allows this kind of fun! it is like a day on the beach!"
# set some variables
go_surf,  get_a_tan_while,  boat_size,  toll_money  =  (
1,'windsurfing', 40.0, -2.00 )
```

Multiple Statement Groups as Suites (:)

Groups of individual statements making up a single code block are called "suites" in Python. *Compound or complex statements*, such as **if**, **while**, **def**, and **class**, are those which require a header line and a suite. Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. We will refer to the combination of a header line and a suite_o as a *clause*.

Suites Delimited via Indentation

Python employs indentation as a means of delimiting blocks of code. Codes at inner levels are indented via spaces or TABs. Indentation requires exact indentation, in other words, all the lines of code in a suite must be indented at the exact same level. Indented lines starting at different positions or column numbers are not allowed; each line would be considered part of another suite and would more than likely result in syntax errors.

Multiple Statements on a Single Line (;)

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

Example:

Importing a class extends two double line using semicolon

```
import sys; x = 'foo';  
sys.stdout.write(x + '\n')
```

which is similar to importing statement

```
import sys; x = 'foo' sys.stdout.write(x + '\n')
```

caution the reader to be wary of the amount of usage of chaining multiple statements on individual lines since it makes code much less readable. You decide:

```
import sys  
x = 'foo'  
sys.stdout.write(x + '\n')
```

In our example, separating the code to individual lines makes for remarkably improved readerfriendliness.

Modules

1. Each Python script is considered a *module*.
2. Modules have a physical presence as disk files.
3. When a module gets large enough or has diverse enough functionality, it may make sense to move some of the code out to another module.
4. Code that resides in modules may belong to an application (i.e., a script that is directly executed), or may be executable code in a library-type module that may be "imported" from another module for invocation.

Variable Assignment

This section describes variable assignment and about the identifiers which make valid variables.

Equal sign (=) is the assignment operator

The equal sign (=) is the main Python assignment operator

Example of variable assignment

```
anInt = -12  
String = 'cart'
```

This assignment does not explicitly assign a value to a variable, although it may appear that way from your experience with other programming languages. In Python, objects are referenced, so on assignment, a reference (not a value) to an object is what is being assigned, whether the object was just created or was a pre-existing object.

Statements such as the following are invalid in Python:

```
>>> x = 1
>>> y = (x = x + 1) # assignments not expressions!
File "<stdin>", line 1
y = (x = x + 1)
^
SyntaxError: invalid syntax
```

Beginning in Python 2.0, the equals sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as *augmented assignment*, statements such as

```
x = x + 1
can now be written as
x += 1
```

Python does not support pre-/post-increment nor pre-/post-decrement operators such as `x++` or `--x`.

How To Do a Multiple Assignment

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

In the above example, an integer object (with the value 1) is created, and x, y, and z are all assigned the same reference to that object. This is the process of assigning a single object to multiple variables. It is also possible in Python to assign multiple objects to multiple variables.

How to Do a "Multiple" Assignment

Another way of assigning multiple variables is using what we shall call the "multiple" assignment. This is not an official Python term, but we use "multiple" here because when assigning variables this way, the objects on both sides of the equals sign are tuples,

```
>>> x, y, z = 1, 2, 'a string'
>>> x
1
>>> y
2
>>> z
'a string'
```

In the above example, two integer objects (with values 1 and 2) and one string object are assigned to x, y, and z respectively. Parentheses are normally used to denote tuples, and although they are optional,

we recommend them anywhere they make the code easier to read:

```
>>> (x, y, z) = (1, 2, 'a string')
```

If you have ever needed to swap values in other languages like C, you will be reminded that a temporary variable, i.e., tmp, is required to hold one value which the other is being exchanged:

```
/* swapping variables in C */  
tmp = x;  
x = y;  
y = tmp;
```

In the above C code fragment, the values of the variables x and y are being exchanged. The tmp variable is needed to hold the value of one of the variables while the other is being copied into it. After that step, the original value kept in the temporary variable can be assigned to the second variable. One interesting side effect of Python's "multiple" assignment is that we no longer need a temporary variable to swap the values of two variables.

```
# swapping variables in Python  
>>> (x, y) = (1, 2)  
>>> x  
1  
>>> y  
2  
>>> (x, y) = (y, x)  
>>> x  
2  
>>> y
```

Identifiers

Identifiers are the set of valid strings which are allowed as names in a computer language. From this all-encompassing list, we segregate out those which are *keywords*, names that form a construct of the language. Such identifiers are reserved words which may not be used for any other purpose, or else a syntax error (Syntax Error exception) will occur. Python also has an additional set of identifiers known as *built-ins*, and although they are not reserved words, use of these special names is not recommended.

Valid Python Identifiers

The rules for Python identifier strings are not unlike most other high-level programming languages:

- First character must be letter or underscore (_)

- Any additional characters can be alphanumeric or underscore

- Case-sensitive

- No identifiers can begin with a number, and no symbols other than the underscore are ever allowed.

Keywords

Python currently has twenty-eight keywords. Generally, the keywords in any language should remain relatively stable, but should things ever change (as Python is a growing and evolving language), a list of keywords as well as an `iskeyword()` function are available in the keyword module.

**Break,except,import,print,class,exec,in,raise,continue,finally,is,return,def,for,lamb
da,try,del,from, and, elif, global, or, assert, else, if, pass, not and while**
These are the keywords of python language

Built-ins

In addition to keywords, Python has a set of "built-in" names which are either set and/or used by the interpreter that are available at any level of Python code. Although not keywords, built-ins should be treated as "reserved for the system" and not used for any other purpose. However, some circumstances may call for *overriding* (a.k.a. redefining, replacing) them. Python does not support overloading of identifiers, so only one name "binding" may exist at any given time.

Special Underscore Identifiers

Python designates (even more) special variables with underscores both prefixed and suffixed. We will also discover later that some are quite useful to the programmer while others are unknown or useless. Here is a summary of the special underscore usage in Python:

- _xxx do not import with '**from module import ***'
- _xxx__ system-defined name
- _xxx request private name mangling in classes

Basic Style Guidelines

Comments

You do not need to be reminded that comments are useful both to you and those who come after you. This is especially true for code that has been untouched by man (or woman) for a time (that means several months in software development time). Comments should not be absent, nor there novellas. Keep the comments explanatory, clear, short, and concise, but get them in *there*. In the end, it saves time and energy for everyone.

Documentation

Python also provides a mechanism whereby documentation strings can be retrieved dynamically through the `__doc__` special variable. The first unassigned string in a module, class declaration, or function declaration can be accessed through by using `obj.__doc__` where *obj* is the module, class, or function name.

Indentation

Since indentation plays a major role, you will have to decide on a spacing style that is easy to read as well as the least confusing. Common sense also plays a recurring rôle in choosing how many spaces or columns to indent.

1 or 2 probably not enough; difficult to determine which block of code statements belong to

8 to 10 may be too many; code which has many embedded levels will wraparound, causing the source to be difficult to read

Four (4) spaces is very popular, not to mention being the preferred choice of Python's creator. Five (5) and six (6) are not bad, but text editors usually do *not* use these settings, so they are not as commonly used. Three (3) and seven (7) are borderline cases. As far as TABs go, bear in mind that different text editors have different concepts of what TABs are. It is advised not to use TABs if your code will live and run on different systems or be accessed with different text editors.

Choosing Identifier Names

Although variable length is no longer an issue with programming languages of today, it is still a good idea to keep name sizes reasonable. The same applies for naming your modules (Python files).

Module Structure and Layout

Modules are simply physical ways of logically organizing all your Python code. Within each file, you should set up a consistent and easy-to-read structure. One such layout is the following:

- (1) startup line (Unix) `#usr/bin/env python`
- (2) module documentation “this is test module”
- (3) module imports `import sys`
- (4) variable declarations `debug=1`
- (5) class declarations

```
class fooclass:
    “foo Class”
    pass
```
- (6) function declarations

```
Def test();
    “test function”
    foo=fooclass();
    if debug:
        print ‘ran test’
```

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

(1) Startup line (Unix)

```
"this is a test module"
```

(2) Module documentation

```
import sys  
import string
```

(3) Module imports

```
debug = 1
```

(4) (Global) Variable declarations

```
class FooClass:  
    "Foo class"  
    pass
```

(5) Class declarations (if any)

```
def test():  
    "test function"  
    foo = FooClass()  
    if debug:  
        print 'ran test()'
```

(6) Function declarations (if any)

```
if __name__ == '__main__':  
    test()
```

(7) "main" body

Python uses the object model abstraction for data storage. Any construct which contains any type of value is an object. Although Python is classified as an "object-oriented programming language". OOP is not required to create perfectly working Python applications. You can certainly write a useful Python script without the use of classes and instances. However, Python's object syntax and architecture certainly encourage or "provoke" this type of behavior. Let us now take a closer look at what a Python "object" is. All Python objects have the following three characteristics:

an identity, a type, and a value.

IDENTITY- Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the `id()` built-in function. This value is as close as you will get to a "memory address" in Python (probably much to the relief of some of you). Even better is that you rarely, if ever, access this value, much less care what it is at all.

TYPE- An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. You can use the `type()` built-in function to reveal the type of a Python object. Since types are also objects in Python, `type()` actually returns an object to you rather than a simple literal.

VALUE Data item that is represented by an object.

Standard Types

The basic standard data types that python supports are:

1. Numbers (four separate sub-types)
2. Regular or "Plain" Integer
3. Long Integer
4. Floating Point Real Number
5. Complex Number
6. String
7. List
8. Tuple
9. Dictionary

Other Built-in Types:

Other types includes the following

1. None
2. File
3. Function
4. Module
5. Class
6. Class Instance and Method

Code Objects

1. Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the `compile()` built-in function.
2. Such objects are appropriate for execution by either **exec** or by the `eval()` built-in function.
3. Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which *do* contain some execution context. (The actual byte-compiled code as a code object is one attribute belonging to a function).
4. Along with the code object, a function's attributes also consist of the administrative support which a function requires, including its name, documentation string, default arguments, and global namespace.

Frames

1. These are objects representing execution stack frames in Python.
2. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment.
3. Some of its attributes include a link to the previous stack frame, the code object (see above) that is being executed, dictionaries for the local and global namespaces, and the current instruction.
4. Each function call results in a new frame object, and for each frame object, a C stack frame is created as well. One place where you can access a frame object is in a traceback object.

Tracebacks

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

```
Traceback (innermost last): File "<stdin>", line N?, in ???  
ErrorName: error reason
```

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

Slice Objects

Slice objects are created when using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include *stride indexing*, multidimensional indexing, and indexing using the Ellipsis type.

The syntax for multi-dimensional indexing is

```
sequence[start1 : end1, start2 : end2], or using the ellipsis,  
sequence[..., start1 : end1].
```

Slice objects can also be generated by the `slice()` built-in function. Extended slice syntax is currently supported only in external third party modules such as the NumPy module and JPython. Stride indexing for sequence types allows for a third slice element that allows for "step"- like access with a syntax of

```
sequence[starting_index : ending_index : stride].
```


We will demonstrate an example of stride indexing using JPython here:

```
>>> foostr = 'abcde'
>>> foostr[::-1] output 'edcba'
>>> foostr[::-2] output 'eca'
>>> foolist = [123, 'xba', 342.23, 'abc']
>>> foolist[::-1] output ['abc', 342.23, 'xba', 123]
```

Ellipsis

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax (...). Like the Null object, ellipsis objects also have a single name, Ellipsis, and has a Boolean *true* value at all times.

Xranges

XRange objects are created by the built-in function xrange(), a sibling of the range() built-in function and used when memory is limited and for when range() generates an unusually large data set

Standard Type Operators

Value Comparison

Comparison operators are used to determine equality of two data values between members of the same type. These comparison operators are supported for all built-in types.

Comparisons yield true or false values, based on the validity of the comparison expression. Python chooses to interpret these values as the plain integers 0 and 1 for false and true, respectively, meaning that each comparison will result in one of those two possible values

S.No	Operator	Function
1	$expr1 < expr2$	<i>expr1 is less than expr2</i>
2	$expr1 > expr2$	<i>expr1 is greater than expr2</i>
3	$expr1 \leq expr2$	<i>expr1 is less than or equal to expr2</i>
4	$expr1 \geq expr2$	<i>expr1 is greater than or equal to expr2</i>
5	$expr1 == expr2$	<i>expr1 is equal to expr2</i>
6	$expr1 != expr2$	<i>expr1 is not equal to expr2 (C-style)</i>
7	$expr1 <> expr2$	<i>expr1 is not equal to expr2</i>

Object Identity Comparison

In addition to value comparisons, Python also supports the notion of directly *comparing objects* themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

<i>S.no</i>	<i>Operator</i>	<i>Function</i>
<i>1</i>	<i>obj1 is obj2</i>	<i>obj1 is the same object as obj2</i>
<i>2</i>	<i>obj1 is not obj2</i>	<i>obj1 is not the same object as obj2</i>

NUMBERS:

Python has four numeric types:

- 1. regular or "plain" integers,
- 2. long integers,
- 3. floating point real numbers, and
- 4. complex numbers.

Numbers provide literal or scalar storage and direct access. Numbers are also an immutable type, meaning that changing or updating its value results in a newly allocated object. This activity is, of course, transparent to both the programmer and the user, so it should not change the way the application is developed.

How to Create and Assign Numbers (Number Objects)

Creating numbers is as simple as assigning a value to a variable.

```
anInt = 1
```

```
1aLong = -999999999999999999L
```

How to Update Numbers

You can "update" an existing number by (re)assigning a variable to another number. The new value can be related to its previous value or to a completely different number altogether.

```
anInt = anInt + 1, aFloat = 2.718281828
```

How to Remove Numbers

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, just use the **del** statement. You can no longer use the variable name, once removed, unless you assign it to a new object; otherwise, you will cause a `NameError` exception to occur.

```
del anInt
```

```
del aLong, aFloat, aComplex
```

Integers

Python has two types of integers. Plain integers are the generic vanilla (32-bit) integers recognized on most systems today. Python also has a long integer size; however, these far exceed the size provided by C longs. We will take a look at both types of Python integers, followed by a description of operators and built-in functions applicable only to Python integer types.

(Plain) Integers

Python's "plain" integers are the universal numeric type. Most machines (32-bit) running Python will provide a range of -2^{31} to $2^{31}-1$, that is -2,147,483,648 to 2,147,483,647. Here are some examples of Python integers:

0101 84 -237 0x80 017 -680 -0X92

Python integers are implemented as (signed) longs in C. Integers are normally represented in base 10 decimal format, but they can also be specified in base eight or base sixteen representation. Octal values have a "0" prefix, and hexadecimal values have either "0x" or "0X" prefixes.

Long Integers

The major difference between the long integers in C or other compiled languages and python these values are typically restricted to 32- or 64-bit sizes, whereas Python long integers are limited only by the amount of (virtual) memory in your machine. Long integers are a superset of integers and are useful when the range of plain integers exceeds those of your application, meaning less than -2^{31} or greater than $2^{31}-1$. Use of long integers is denoted by an upper- or lowercase (L) or (l), appended to the integer's numeric value. Values can be expressed in decimal, octal, or hexadecimal. The following are examples of long integers:

16384L -0x4E8L 017L -2147483648l 052144364L

Floating Point Real Numbers

Floats in Python are implemented as C doubles, double precision floating point real numbers, values which can be represented in straight forward decimal or scientific notations. These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent (this gives you about $\pm 10^{308.25}$ in range), and the final bit to the sign. However, the actual amount of precision you will receive (along with the range and overflow handling) depends completely on the architecture of the machine as well as the implementation of the compiler which built your Python interpreter.

Floating point values are denoted by a decimal point (.) in the appropriate place and an optional "e" suffix representing scientific notation. We can use either lowercase (e) or uppercase (E). Positive (+) or negative (-) signs between the "e" and the exponent indicate the sign of the exponent. Absence of such a sign indicates a positive exponent. Here are some floating point values:

```
0.0 -777. 1.6 -5.555567119 96e3 * 1.0
4.3e25 9.384e-23 -2.172818 float(12) 1.0000000001
3.1416 4.2E-10 -90. 6.022e23 -1.609E-19
```

Complex Numbers

A long time ago, mathematicians were stumped by the following equation:

$x^2 = -1$ The reason for this is because any real number (positive or negative) multiplied by itself results in a positive number. How can you multiply any number with itself to get a negative number? No such real number exists. So in the eighteenth century, mathematicians invented something called an *imaginary number* i (or j — depending what math book you are reading) such that: $j = \sqrt{-1}$ Basically a new branch of mathematics was created around this special number (or concept), and now imaginary numbers are used in numerical and mathematical applications. Combining a real number with an imaginary number forms a single entity known as a *complex number*. A complex number is any ordered pair of floating point real numbers (x, y) denoted by $x + yj$ where x is the real part and y is the imaginary part of a complex number.

Here are some facts about Python's support of complex numbers:

1. Imaginary numbers by themselves are not supported in Python
2. Complex numbers are made up of real and imaginary parts
3. Syntax for a complex number: *real+imag j*
4. Both real and imaginary components are floating point values
5. Imaginary part is suffixed with letter "J" lowercase (j) or upper (J)

The following are examples of complex numbers:

64.375+1j 4.23-8.5j 0.23-8.55j 1.23e-045+6.7e+089j
6.23+1.5j -1.23-875J 0+1j9.80665-8. 31441J -.0224+0j

Complex Number Built-in Attributes

Complex numbers are one example of objects with data attributes . The data attributes are the real and imaginary components of the complex number object they belong to. Complex numbers also have a method attribute which can be invoked, returning the complex conjugate of the object.


```
>>> aComplex = -8.333-1.47j
```

```
>>> aComplex  
(-8.333-1.47j)
```

```
>>> aComplex.real  
-8.333
```

```
>>> aComplex.imag  
-1.47
```

```
>>> aComplex.conjugate()  
(-8.333+1.47j)
```

Built-in Functions

Standard Type Functions

we introduced the `cmp()`, `str()`, and `type()` built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and tell you a number's type, respectively. Here are some examples of using these functions:

```
>>> cmp(-6, 2]                o/p -1  
>>> cmp(-4.333333, -2.718281828) o/p -1  
>>> cmp(0xFF, 255)            o/p 0  
>>> str(0xFF)                 o/p 5'
```

Operational

Python has five operational built-in functions for numeric types: `abs()`, `coerce()`, `divmod()`, `pow()`, and `round()`.

Abs()

`abs()` returns the absolute value of the given argument. If the argument is a complex number, then `math.sqrt(num. real2 + num. imag2)` is returned. Here are some examples of using the `abs()` built-in function:

```
>>> abs(-1) o/p 1
>>> abs(10.) o/p 10.0
```

Coercion

The `coerce()` function, although it technically is a numeric type conversion function, does not convert to a specific type and acts more like an operator, hence our placement of it in our operational built-ins section. The `coerce()` function is a way for the programmer to explicitly coerce a pair of numbers rather than letting the interpreter do it. This feature is particularly useful when defining operations for newly-created numeric class types. `coerce()` just returns a tuple containing the converted pair of numbers.

```
>>> coerce(1, 134L) o/p (1L, 134L)
```

Divmod

The `divmod()` built-in function combines division and modulus operations into a single function call that returns the pair (quotient, remainder) as a tuple. The values returned are the same as those given for the standalone division and modulus operators for integer types. For floats, the quotient returned is `math.floor(num1/num2)` and for complex numbers, the quotient is `math.floor((num1/num2).real)`.

```
>>> divmod(10,3) o/p (3, 1)
```

are differences other than the fact that one is an operator and the other is a built-in function.

Pow()

The `**` operator did not appear until Python 1.5, and the `pow()` built-in takes an optional third parameter, a modulus argument. If provided, `pow()` will perform the exponentiation first, then return the result modulo the third argument. This feature is used for cryptographic applications and has better performance than `pow(x,y) % z` since the latter performs the calculations in Python rather than in C like `pow(x, y, z)`.

```
>>> pow(2,5) o/p 32
```

```
>>> pow(5,2) o/p 25
```

Storage Model

- 1. The first way we can categorize the types is by how many objects can be stored in an object of this type.
- 2. Python's types, as well as types from most other languages, can hold either single or multiple values.
- 3. A type which holds a single object we will call *literal* or *scalar* storage, and those which can hold multiple objects we will refer to as *container* storage.
- 4. Container types bring up the additional issue of whether different types of objects can be stored. All of Python's container types can hold objects of different types.

Table 4.6. Types categorized by the Storage Model	
Storage model category	Python types that fit category
literal/scalar	numbers (all numeric types), strings
container	lists, tuples, dictionaries

Update Model

Another way of categorizing the standard types is by asking the question, "Once created, can objects be changed or their values updated?" When we introduced Python types early on, we indicated that certain types allow their values to be updated and others do not. Mutable objects are those whose values can be changed, and immutable objects are those whose values cannot be changed. Table 4.7 illustrates which types support updates and which do not.

Table 4.7. Types Categorized by the Update Model	
Update model category	Python types that fit category
mutable	lists, dictionaries
immutable	numbers, strings, tuples

Access Model

Although the previous two models of categorizing the types are useful when being introduced to Python, they are not the primary models for differentiating the types. For that purpose, we use the access model. By this, we mean, how do we access the values of our stored data? There are three categories under the access model: *direct*, *sequence*, and *mapping*. The different access models and which types fall into each respective category are given in Table 4.8.

Table 4.8. Types Categorized by the Access Model	
access model category	types that fit category
direct	numbers
sequence	strings, lists, tuples
mapping	dictionaries

Round()

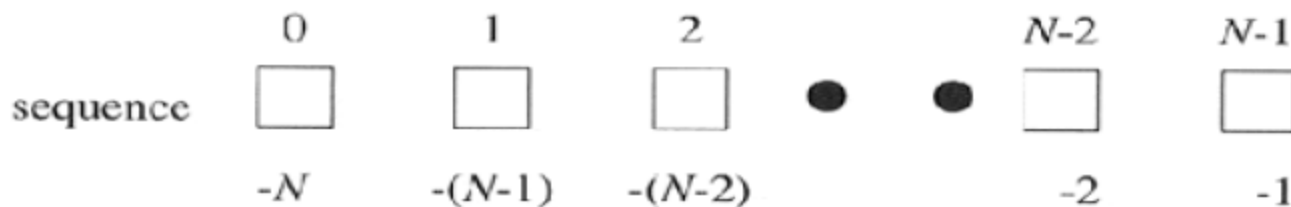
The `round()` built-in function has a syntax of `round (flt,ndig=0)`. It normally rounds a floating point number to the nearest integral number and returns that result (still) as a float. When the optional third `ndig` option is given, `round()` will round the argument to the specific number of decimal places.

```
>>> round(3) o/p 3.0
```

```
>>> round(3.45) o/p 3.0
```

SEQUENCES:

Sequence types all share the same access model: ordered set with sequentially-indexed offsets to get to each element. Multiple elements may be achieved by using the slice operators which we will explore in this chapter. The numbering scheme used starts from zero (0) and ends with one less the length of the sequence—the reason for this is because we began at 0. Figure6-1 illustrates how sequence items are stored. Python's powerful sequence types are strings, lists, and tuples.



$N == \text{length of sequence} == \text{len}(\text{sequence})$

Operators

A list of all the operators applicable to all sequence types is given below. The operators appear in hierarchical order from highest to lowest with the levels alternating between shaded and unshaded.

<i>Sequence Operator</i>	<i>Function</i>
<i>seq[ind]</i>	element located at index <i>ind</i> of <i>seq</i>
<i>seq[ind1:ind2]</i>	elements from index <i>ind1</i> to <i>ind2</i> of <i>seq</i>
<i>seq * expr</i>	<i>seq</i> repeated <i>expr</i> times
<i>seq1 + seq2</i>	concatenates sequences <i>seq1</i> and <i>seq2</i>
<i>obj in seq</i>	tests if <i>obj</i> is a member of sequence <i>seq</i>
<i>obj not in seq</i>	tests if <i>obj</i> is not a member of sequence <i>seq</i>

Membership (in, not in)

Membership test operators are used to determine whether an element is *in* or is a member of a sequence. For strings, this test is whether a character is in a string, and for lists and tuples, it is whether an object is an element of those sequences. The **in** and **not in** operators are Boolean in nature; they return the integer one if the membership is confirmed and zero otherwise. The syntax for using the membership operators is as follows:

obj [**not**] **in** *sequence*

Concatenation (+)

This operation allows us to take one sequence and join it with another sequence of the same type. The syntax for using the concatenation operator is as follows:

sequence1 + sequence2

The resulting expression is a new sequence which contains the combined contents of sequences

sequence1 and *sequence2*.

Repetition (*)

The repetition operator is useful when consecutive copies of sequence elements are desired. The syntax for using the membership operators is as follows:

sequence * copies_int

The number of copies, *copies_int*, must be a plain integer. It cannot even be a long. As with the concatenation operator, the object returned is newly allocated to hold the contents of the multiplyreplicated objects.

Slices ([], [:])

Sequences are structured data types whose elements are placed sequentially in an ordered manner. This format allows for individual element access by index offset or by an index range of indices to "grab" groups of sequential elements in a sequence. This type of access is called *slicing*, and the slicing operators allow us to perform such access. The syntax for accessing an individual element is:

sequence[index]

sequence is the name of the sequence and *index* is the offset into the sequence where the desired element is located. Index values are either positive, ranging from 0 to the length of the sequence less one, i.e., $0 \leq \textit{index} \leq \text{len}(\textit{sequence}) - 1$, or negative, ranging from -1 to the negative length of the sequence, $-\text{len}(\textit{sequence}) \leq \textit{index} \leq -1$. The difference between the positive and negative indexes is that positive indexes start from the beginning of the sequences and negative indexes begin from the end. Accessing a group of elements is similar. Starting and ending indexes may be given, separated by a colon (:). The syntax for accessing a group of elements is:

sequence [[starting_index]: [ending_index]

Using this syntax, we can obtain a "slice" of elements in *sequence* from the *starting_index* up to but not including the element at the *ending_index* index. Both *starting_index* and *ending_index* are optional, and if not provided, the slice will go from the beginning of the sequence or until the end of the sequence, respectively.

Built-in Functions

Conversion

The `list()`, `str()`, and `tuple()` built-in functions are used to convert from any sequence type to another.

Function

`list (seq)`

`str (obj)`

`tuple (seq)`

Operation

converts *seq* to list

converts *obj* to string

converts *seq* to tuple

We use the term "convert" loosely. It does not actually convert the argument object into another type; recall that once Python objects are created, we cannot change their identity or their type. Rather, these functions just create a new sequence of the requested type, populate it with the members of the argument object, and pass that new sequence back as the return value. The `str()` function is most popular when converting an object into something printable and works with other types of objects, not just sequences. The `list()` and `tuple()` functions are useful to convert from one to another (lists to tuples and vice versa). However, although those functions are applicable for strings as well since strings are sequences, using `tuple()` and `list()` to turn strings into tuples or lists is not common practice.

Operational

Python provides the following operational built-in functions for sequence types.

Sequence Type Operational Built-in Functions

<i>Function</i>	<i>Operation</i>
<code>len(seq)</code>	returns length (number of items) of <i>seq</i>
<code>max(seq)</code>	returns "largest" element in <i>seq</i>
<code>min(seq)</code>	returns "smallest" element in <i>seq</i>

Strings:

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. This contrasts with most other scripting languages, which use single quotes for literal strings and double quotes to allow escaping of characters. Python uses the "raw string" operator to create literal quotes, so no differentiation is necessary. Python does not have a character type; this is probably another reason why single and double quotes are the same. Nearly every Python application uses strings in one form or another. Strings are a literal or scalar type, meaning they are treated by the interpreter as a singular value and are not containers which hold other Python objects. Strings are immutable, meaning that changing an element of a string requires creating a new string. Strings are made up of individual characters, and such elements of strings may be accessed sequentially via slicing.

How to Create and Assign Strings

Creating strings is as simple as assigning a value to a variable:

```
>>> aString = 'Hello World!'
```

```
>>> print aString  
Hello World!
```

How to Access Values(Characters and Substrings) in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

How to Update Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

Like numbers, strings are not mutable, so you cannot change an existing string without creating a new one from scratch. That means that you cannot update individual characters or substrings in a string. However, as you can see above, there is nothing wrong with piecing together part of your old string and assigning it to a new string. ⁴⁵

How to Remove Characters and Strings

To repeat what we just said, strings are immutable, so you cannot remove individual characters from an existing string. What you can do, however, is to empty the string, or to put together another string which drops the pieces you were not interested in. Let us say you want to remove one letter from "Hello World!"... the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString
'Helo World!'
```

To clear or remove a string, you assign an empty string or use the **del** statement, respectively:

```
>>> aString = ""
>>> aString
"
```

```
>>> del aString
```

In most applications, strings do not need to be explicitly deleted. Rather, the code defining the string eventually terminates, and the string is automatically garbage-collected.

Strings and Operators

Standard Type Operators

We introduced a number of operators that apply to most objects, including the standard types. We will take a look at how some of those apply to strings. For a brief introduction, here are a few examples using strings:

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> str1 < str2
1
>>> str2 != str3
1
>>> (str1 < str3) and (str2 == 'xyz')
0
```

When using the value comparison operators, strings are compared lexicographically (ASCII value order).

Sequence Operators

Slices ([] and [:])

We will apply that knowledge to strings in this section. In particular, we will look at:

- Counting forward
- Counting backward
- Default/missing indexes

For the following examples, we use the single string 'abcd'. Provided in the figure is a list of positive and negative indexes that indicate the position in which each character is located within the string itself. Using the length operator, we can confirm that its length is 4:

```
>>> string = 'abcd'
>>> len(string)
4
```

When counting forward, indexes start at 0 to the left and end at one less than the length of the string (because we started from zero). In our example, the final index of our string is $\text{final index} = \text{len(string)} - 1 = 4 - 1 = 3$

We can access any substring within this range. The slice operator with a single argument will give us a single character, and the slice operator with a range, i.e., using a colon (:), will give us multiple consecutive characters. Again, for any ranges [*start:end*], we will get all characters starting at offset *start* up to, but not including, the character at *end*. In other words, for all characters *x* in the range [*start : end*], $\text{start} \leq x < \text{end}$.

```
>>> string[0]
'a'
>>> string[1:3]
'bc'
>>> string[2:4]
'cd'
>>> string[4]
```

Traceback (innermost last):

File "<stdin>", line 1, in ?

IndexError: string index out of range

Any index outside our valid index range (in our example, 0 to 3) results in an error. Above, our access of `string[2:4]` was valid because that returns characters at indexes 2 and 3, i.e., 'c' and 'd', but a direct access to the character at index 4 was invalid. When counting backward, we start at index -1 and move toward the beginning of the string, ending at negative value of the length of the string. The final index (the first character) is located at:

```
final index = -len(string)
= -4
>>> string[-1]
'd'
>>> string[-3:-1]
'bc'
>>> string[-4]
'a'
```

When either a starting or an ending index is missing, they default to the beginning or end of the string, respectively.

```
>>> string[2:]
'cd'
>>> string[1:]
'bcd'
>>> string[:-1]
'abc'
>>> string[:]
'abcd'
```

Membership (in, not in)

The membership question asks whether a character (string of length one) appears in a string. A one is returned if that character appears in the string and zero otherwise. Note that the membership operation is not used to determine if a substring is within a string. Such functionality can be accomplished by using the string methods or string module functions `find()` or `index()` (and their brethren `rfind()` and `rindex()`). Here are a few more examples of strings and the membership operators.

```
>>> 'c' in 'abcd'
1
>>> 'n' in 'abcd'
0
>>> 'n' not in 'abcd'
1
```

Concatenation (+)

We can use the concatenation operator to create new strings from existing ones. We have already seen the concatenation operator. Here are a few more examples:

```
>>> 'Spanish' + 'Inquisition'
'SpanishInquisition'
>>> s = 'Spanish' + ' ' + 'Inquisition' + ' Made Easy'
>>> s
```

Spanish Inquisition Made Easy'

Repetition (*)

The repetition operator creates new strings, concatenating multiple copies of the same string to accomplish its functionality:

```
>>> 'Ni!' * 3
'Ni!Ni!Ni!'
>>>
>>> print '-' * 20, 'Hello World!', '-' * 20
----- Hello World! -----
```

String-only Operators

Format Operator (%)

One of Python's coolest features is the string format operator. This operator is unique to strings and makes up for the lack of having functions from C's `printf()` family. In fact, it even uses the same symbol, the percent sign (%), and supports all the `printf()` formatting codes. The syntax for using the format operator is as follows:

format_string % (*arguments_to_convert*)

The *format_string* on the left-hand side is what you would typically find as the first argument to `printf()`, the format string with any of the embedded % codes. The *arguments_to_convert* parameter matches the remaining arguments you would send to `printf()`, namely the set of variables to convert and display.

Format Symbol

Conversion

<code>%c</code>	character
<code>%s</code>	string conversion via <code>str()</code> prior to formatting
<code>%i</code>	signed decimal integer
<code>%d</code>	signed decimal integer
<code>%u</code>	unsigned decimal integer
<code>%o</code>	octal integer
<code>%x</code>	hexadecimal integer (lowercase letters)
<code>%X</code>	hexadecimal integer (UPPERcase letters)
<code>%e</code>	exponential notation (with lowercase 'e')
<code>%E</code>	exponential notation (with UPPERcase 'E')
<code>%f</code>	floating point real number
<code>%g</code>	the shorter of <code>%f</code> and <code>%e</code>
<code>%G</code>	the shorter of <code>%f</code> and <code>%E</code>

Python supports two formats for the input arguments. The first is a tuple, which is basically the set of arguments to convert, just like for C's `printf()`. The second format which Python supports is a dictionary. A dictionary is basically a set of hashed key-value pairs. The keys are requested in the *format_string*, and the corresponding values are provided when the string is formatted. Converted strings can either be used in conjunction with the **print** statement to display out to the user or saved into a new string for future processing or displaying to a graphical user interface. Other supported symbols and functionality are

<i>Symbol</i>	<i>Functionality</i>
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	<i>m</i> is the minimum total width and <i>n</i> is the number of digits to display after the decimal point (if appl.)

As with C's `printf()`, the asterisk symbol (*) may be used to dynamically indicate the width and precision via a value in argument tuple. Before we get to our examples, one more word of caution: long integers are more than likely too large for conversion to standard integers, so we recommend using exponential notation to get them to fit. Here are some examples using the string format operator:

Hexadecimal Output

```
>>> "%x" % 108 >>> "%X" % 108
'6c' '6C'
>>>
```

Floating Point and Exponential Notation Output

```
>>>
>>> '%f' % 1234.567890
'1234.567890'
>>>
>>> '%.2f' % 1234.567890
'1234.57'
>>>
```

Integer and String Output

```
>>> "%+d" % 4
'+4'
>>>
>>> "%+d" % -4
'-4'
>>>
>>> "we are at %d%%" % 100
```

Lists:

Like strings, lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there. Strings consist only of characters and are immutable (cannot change individual elements) while lists are flexible container objects which hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy, too, as we see in the following examples. The objects that you can place in a list can include standard types and objects as well as user-defined ones. Lists can contain different types of objects and are more flexible than an array of C structs or Python arrays (available through the external array module) because arrays are restricted to containing objects of a single type. Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will. Tuples share many of the same characteristics of lists and although we have a separate section on tuples, many of the examples and list functions are applicable to tuples as well. The key difference is that tuples are immutable, i.e., read-only, so any operators or functions which allow updating lists, such as using the slice operator on the left-hand side of an assignment, will not be valid for tuples.

How to Create and Assign Lists

Creating lists is as simple as assigning a value to a variable. You handcraft a list (empty or with elements) and perform the assignment. Lists are delimited by surrounding square brackets

(`[]`).

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
>>> anotherList = [None, 'something to see here']
>>> print aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
```

How to Access Values in Lists

Slicing works similar to strings; use the square bracket slice operator (`[]`) along with the index or indices.

```
>>> aList[0]
123
>>> aList[1:4]
['abc', 4.56, ['inner', 'list']]
```

How to Update Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method:

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList.append(float replacer)
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j), float replacer]
```

How to Remove List Elements and Lists

To remove a list element, you can use either the **`del`** statement if you know exactly which element(s) you are deleting or the `remove()` method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del aList[1]
```



```
>>> aList
```

```
[123, 'float replacer', ['inner', 'list'], (7-9j)]
```

You can also use the `pop()` method to remove and return a specific object from a list. Normally, removing an entire list is not something application programmers do. Rather, they tend to let it go out of scope (i.e., program termination, function call completion, etc.) and be garbagecollected, but if they do want to explicitly remove an entire list, use the **del** statement: **del aList**

Sequence Type Operators

Slices ([] and [:])

Slicing with lists is very similar to strings, but rather than using individual characters or substrings, slices of lists pull out an object or a group of objects which are elements of the list operated on. Focusing specifically on lists, we make the following definitions:

```
>>> num_list = [43, -1.23, -2, 6.19e5]
```

```
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
```

```
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
```

Slicing operators obey the same rules regarding positive and negative indexes, starting and ending indexes, as well as missing indexes, which default to the beginning or to the end of a sequence.

Membership (in, not in)

With strings, the membership operator determined whether a single character is a member of a string. With lists (and tuples), we can check whether an object is a member of a list (or tuple).

```
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> 'beef' in mixup_list
1
```

Concatenation (+)

The concatenation operator allows us to join multiple lists together. Note in the examples below that there is a restriction of concatenating like objects. In other words, you can concatenate only objects of the same type. You cannot concatenate two different types even if both are sequences.

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> str_list + num_list
['jack', 'jumped', 'over', 'candlestick', 'park', 43, -1.23, -2, 619000.0]
```

Repetition (*)

Use of the repetition operator may make more sense with strings, but as a sequence type, lists and tuples can also benefit from this operation, if needed:

```
>>> num_list * 2
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
>>>
>>> num_list * 3
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0, 43, -1.23, -2,
619000.0]
```

Other functions are Cmp, len, max, and min.

Cmp() Function

```
>>> str1 = 'abc'
>>> str2 = 'lmn'
>>> str3 = 'xyz'
>>> cmp(str1, str2)
-11
```

len() Function

```
>>> str1 = 'abc'
>>> len(str1)
3
```

max() and min() Function

max() and min() did not have a significant amount of usage for strings since all they did was to find the "largest" and "smallest" characters (lexicographically) in the string **list()** and **tuple()**. The list() and tuple() methods take sequence types and convert them to lists and tuples, respectively. Although strings are also sequence types, they are not commonly used with list() and tuple(). These built-in functions are used more often to convert from one type to the other., i.e., when you have a tuple that you need to make a list (so that you can modify its elements) and vice versa.

List Type Built-in Methods

List Method

Operation

list.append(obj)

appends object *obj* to *list*

list.count(obj)

returns count of how many times *obj* occurs in *list*

list.extend(seq)[a]

appends the contents of *seq* to *list*

list.index(obj)

returns the lowest index in *list* that *obj* appears

list.insert(index, obj)

inserts object *obj* into *list* at offset *index*

list.pop(obj=list[-1])[a]

removes and returns last object or *obj* from *list*

list.remove(obj)

removes object *obj* from *list*

list.reverse()

reverses objects of *list* in place

list.sort([func])

sorts objects of list, use compare *func* if given

```
>>> a = [10,20,30,40,50]
>>> a
[10, 20, 30, 40, 50]
>>> a.append(60)
>>> a
[10, 20, 30, 40, 50, 60]
>>> a.append(70)
>>> a
[10, 20, 30, 40, 50, 60, 70]
>>> a.append(70)
>>> a
[10, 20, 30, 40, 50, 60, 70, 70]
>>> a.count(70)
2
>>> a.index(40)
3
>>> a.insert(2,1000)
>>> a
[10, 20, 1000, 30, 40, 50, 60, 70, 70]
>>> a.pop()
70
>>> a
[10, 20, 1000, 30, 40, 50, 60, 70]
>>> a.remove(20)
```

```
>>> a
[10, 1000, 30, 40, 50, 60, 70]
>>> a.reverse()
>>> a
[70, 60, 50, 40, 30, 1000, 10]
>>> a.sort()
>>> a
[10, 30, 40, 50, 60, 70, 1000]
>>>
```

Tuples:

Tuples are another container type extremely similar in nature to lists. The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets. Functionally there is a more significant difference, and that is the fact that tuples are immutable. Our usual *modus operandi* is to present the operators and built-in functions for the more general objects, followed by those for sequences and conclude with those applicable only for tuples, but because tuples share so many characteristics with lists, we would be duplicating much of our description from the previous section. Rather than providing much repeated information, we will differentiate tuples from lists as they apply to each set of operators and functionality, then discuss immutability and other features unique to tuples.

How to Create and Assign Tuples

Creating and assigning lists are practically identical to lists, with the exception of empty tuples. These require a trailing comma (,) enclosed in the tuple delimiting parentheses (()).

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
```

How to Access Values in Tuples

Slicing works similar to lists: Use the square bracket slice operator ([]) along with the index or indices.

```
>>> aTuple>>> aList[1:4]
('abc', 4.56, ['inner', 'tuple'])
>>> aTuple[:3]
(123, 'abc', 4.56)
>>> aTuple[3][1] 'tuple'
```

How to Update Tuples

Like numbers and strings, tuples are immutable which means you cannot update them or change values of tuple elements. In Sections 6.2 and 6.3.2, we were able to take portions of an existing string to create a new string. The same applies for tuples.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

How to Remove Tuple Elements and Tuples

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded. To explicitly remove an entire list, just use the **del** statement:

```
del aTuple
```

Tuple Operators and Built-in Functions

Standard and Sequence Type Operators and Built-in Functions

Object and sequence operators and built-in functions act the exact same way toward tuples as they do with lists. You can still take slices of tuples, concatenate and make multiple copies of tuples, validate membership, and compare tuples:

Creation, Repetition, Concatenation

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t * 2
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
>>> t = t + ('free', 'easy')
>>> t
(['xyz', 123], 23, -103.4, 'free', 'easy')
```


Membership, Slicing

```
>>> 23 in t
1
>>> 123 in t
0
>>> t[0][1]
123
>>> t[1:]
(23, -103.4, 'free', 'easy')
```

Built-in Functions

```
>>> str(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
>>> len(t)
5
>>> max(t)
'free'
>>> min(t)
-103.4
>>> cmp(t, (['xyz', 123], 23, -103.4, 'free',
             'easy'))
0
>>> list(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

Operators

```
>>> (4, 2) < (3, 5)
0
>>> (2, 4) < (3, -1)
1
>>> (2, 4) == (3, -1)
0
>>> (2, 4) == (2, 4)
1
```

4.4.4 Dictionaries:

Dictionaries are Python's mapping or hashing type. A dictionary is mutable and is another container type that can store any number of Python objects, including other container types. Python dictionaries are implemented as resizable hash tables.

How to Create and Assign Dictionaries

Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

```
>>> dict1 = {}  
>>> dict2 = {'name': 'earth', 'port': 80}  
>>> dict1, dict2  
({}, {'port': 80, 'name': 'earth'})
```

How to Access Values in Dictionaries

To access dictionary elements, you use the familiar square brackets along with the key to obtain its value:

```
>>> dict2['name']  
'earth'  
>>>  
>>> print 'host %s is running on port %d' % \  
... (dict2['name'], dict2['port'])  
host earth is running on port 80
```

Dictionary dict1 is empty while dict2 has two data items. The keys in dict2 are 'name' and 'port', and their associated value items are 'earth' and 80, respectively. Access to the value is through the key, as you can see from the explicit access to the 'name' key. If we attempt to access a data item with a key which is not part of the dictionary, we get an error:

```
>>> dict2['server']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: server
```

In this example, we tried to access a value with the key 'server' which, as you know, does not exist from the code above. The best way to check if a dictionary has a specific key is to use the dictionary's `has_key()` method. We will introduce all of a dictionary's methods below. The Boolean `has_key()` method will return a 1 if a dictionary has that key and 0 otherwise.

```
>>> dict2.has_key('server')
0
>>> dict2.has_key('name')
1
>>> dict2['name']
'earth'
```

How to Update Dictionaries

You can update a dictionary by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry (see below for more details on removing an entry).

```
>>> dict2['name'] = 'venus' # update existing entry
>>> dict2['port'] = 6969 # update existing entry
>>> dict2['arch'] = 'sunos5' # add new entry
>>>
>>> print 'host %(name)s is running on port %(port)d' % dict2
host venus is running on port 6969
```

If the key does exist, then its previous value will be overridden by its new value. The **print** statement above illustrates an alternative way of using the string format operator (%), specific to dictionaries. Using the dictionary argument, you can shorten the print request somewhat because naming of the dictionary occurs only once, as opposed to occurring for each element using a tuple argument. You may also add the contents of an entire dictionary to another dictionary by using the `update()` built-in method.

How to Remove Dictionary Elements and Dictionaries

Removing an entire dictionary is not a typical operation. Generally, you either remove individual dictionary elements or clear the entire contents of a dictionary. However, if you really want to "remove" an entire dictionary, use the **del** statement. Here are some deletion examples for dictionaries and dictionary elements:

```
del dict1['name'] # remove entry with key 'name'
dict1.clear() # remove all entries in dict1
del dict1 # delete entire dictionary
```

Operators

Dictionaries do not support sequence operations such as concatenation and repetition, although an `update()` built-in method exists that populates one dictionary with the contents of another. Dictionaries do not have a "membership" operator either, but the `has_key()` built-in method basically performs the same task.

Standard Type Operators

Dictionaries will work with all of the standard type operators.

>>> dict4 = { 'abc': 123 }	>>> dict5 = { 'abc': 456 }
>>> dict6 = { 'abc': 123, 98.6: 37 }	>>> dict7 = { 'xyz': 123 }
>>> dict4 < dict5 o/p 1	>>> (dict4 < dict6) and (dict4 < dict7) o/p 1
>>> (dict5 < dict6) and (dict5 < dict7) o/p 1	>>> dict6 < dict7 o/p 0

How are all these comparisons performed? Like lists and tuples, the process is a bit more complex than it is for numbers and strings.

Dictionary Key-lookup Operator ([])

The only operator specific to dictionaries is the key-lookup operator, which works very similar to the single element slice operator for sequence types. For sequence types, an index offset is the sole argument or subscript to access a single element of a sequence. For a dictionary, lookups are by key, so that is the argument rather than an index. The keylookup operator is used for both assigning values to and retrieving values from a dictionary:

```
dict[k] = v # set value 'v' in dictionary with key 'k'  
dict[k] # lookup value in dictionary with key 'k'
```

Built-in Functions

Standard Type Functions [type(), str(), and cmp()]

The type() built-in function, when operated on a dictionary, reveals an object of the dictionary type. The str() built-in function will produce a printable string representation of a dictionary. These are fairly straightforward.

In each of the last three chapters, we showed how the cmp() built-in function worked with numbers, strings, lists, and tuples. So how about dictionaries? Comparisons of dictionaries are based on an algorithm which starts with sizes first, then keys, and finally values. In our example below, we create two dictionaries and compare them, then slowly modify the dictionaries to show how these changes affect their comparisons:

```
>>> dict1 = {}  
>>> dict2 = { 'host': 'earth', 'port': 80 }  
>>> cmp(dict1, dict2)  
-1  
>>> dict1['host'] = 'earth'  
>>> cmp(dict1, dict2)  
-1
```

In the first comparison, dict1 is deemed smaller because dict2 has more elements (2 items vs. 0 items). After adding one element to dict1, it is still smaller (2 vs. 1), even if the item added is also in dict2.

```
>>> dict1['port'] = 8080
```

```
>>> cmp(dict1, dict2)
1
>>> dict1['port'] = 80
>>> cmp(dict1, dict2)
0
```

After we add the second element to dict1, both dictionaries have the same size, so their keys are then compared. At this juncture, both sets of keys match, so comparison proceeds to checking their values. The values for the 'host' keys are the same, but when we get to the 'port' key, dict2 is deemed larger because its value is greater than that of dict1's 'port' key (8080 vs. 80). When resetting dict2's 'port' key to the same value as dict1's 'port' key, then both dictionaries form equals: They have the same size, their keys match, and so do their values, hence the reason that 0 is returned by cmp().

```
>>> dict1['prot'] = 'tcp'
>>> cmp(dict1, dict2)
1
>>> dict2['prot'] = 'udp'
>>> cmp(dict1, dict2)
-1
```

As soon as an element is added to one of the dictionaries, it immediately becomes the "larger one," as in this case with dict1. Adding another key-value pair to dict2 can tip the scales again, as both dictionaries' sizes match and comparison progresses to checking keys and values.


```

>>> cdict = { 'fruits':1 }
>>> ddict = { 'fruits':1 }
>>> cmp(cdict, ddict)
0
>>> cdict['oranges'] = 0
>>> ddict['apples'] = 0
>>> cmp(cdict, ddict)
14

```

Our final example reminds us that `cmp()` may return values other than -1, 0, or 1. The algorithm pursues comparisons in the following order:

(1) Compares Dictionary Sizes

If the dictionary lengths are different, then for `cmp(dict1, dict2)`, `cmp()` will return a positive number if *dict1* is longer and a negative number if *dict2* is longer. In other words, the dictionary with more keys is greater, i.e.,

$$\text{len}(\text{dict1}) > \text{len}(\text{dict2}) \text{ ? } \text{dict1} > \text{dict2}$$

(2) Compares Dictionary Keys

If both dictionaries are the same size, then their keys are compared; the order in which the keys are checked is the same order as returned by the `keys()` method. (It is important to note here that keys which are the same will map to the same locations in the hash table. This keeps key-checking consistent.) At the point where keys from both do not match, they are directly compared and `cmp()` will return a positive number if the first differing key for *dict1* is greater than the first differing key of *dict2*.

(3) Compares Dictionary Values

If both dictionary lengths are the same and the keys match exactly, the values for each key in both dictionaries are compared. Once the first key with non-matching values is found, those values are compared directly. Then `cmp()` will return a positive number if, using the same key, the value in *dict1* is greater than that of the value in *dict2*.

(4) Exact Match

If we have reached this point, i.e., the dictionaries have the same length, the same keys, and the same values for each key, then the dictionaries are an exact match and 0 is returned.

Mapping Type Function [`len()`]

Similar to the sequence type built-in function, the mapping type `len()` built-in returns the total number of items, that is, key-value pairs, in a dictionary:

```
>>> dict2 = { 'name': 'earth', 'port': 80 }
>>> dict2
{'port': 80, 'name': 'earth'}
>>> len(dict2)
2
```

We mentioned earlier that dictionary items are unordered. We can see that above, when referencing `dict2`, the items are listed in reverse order from which they were entered into the dictionary.

Built-in Methods

Table 7.1. Dictionary Type Methods	
<i>dictionary method</i>	<i>Operation</i>
<code>dict.clear</code> ^[a] ()	removes all elements of dictionary <i>dict</i>
<code>dict.copy</code> ^[a] ()	returns a (shallow ^[b]) copy of dictionary <i>dict</i>
<code>dict.get</code> (<i>key</i> , <i>default</i> =None) ^[a]	for key <i>key</i> , returns value or <i>default</i> if <i>key</i> not in dictionary (note that <i>default</i> 's default is <i>None</i>)
<code>dict.has_key</code> (<i>key</i>)	returns 1 if <i>key</i> in dictionary <i>dict</i> , 0 otherwise
<code>dict.items</code> ()	returns a list of <i>dict</i> 's (key, value) tuple pairs
<code>dict.keys</code> ()	returns list of dictionary <i>dict</i> 's keys
<code>dict.setdefault</code> <i>key</i> , <i>default</i> =None) ^[c]	similar to <code>get()</code> , but will set <code>dict[key]=default</code> if <i>key</i> is not already in <i>dict</i>
<code>dict.update</code> (<i>dict2</i>) ^[a]	adds dictionary <i>dict2</i> 's key-values pairs to <i>dict</i>
<code>dict.values</code> ()	returns list of dictionary <i>dict</i> 's values

```
>>> dict2 = { 'name': 'earth', 'port': 80 }
>>> dict2.has_key('name')
1
>>> dict2['name']
'earth'
>>> dict2.has_key('number')
0
```

```

>>> dict2.keys()
['port', 'name']
>>> dict2.values()
[80, 'earth']
>>> dict2.items()
[('port', 80), ('name', 'earth')]
>>> for eachKey in dict2.keys():
... print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
...
dict2 key port has value 80
dict2 key name has value earth
>>> dict2Keys = dict2.keys()
>>> dict2Keys.sort()
>>> for eachKey in dict2Keys:
... print 'dict2 key', eachKey, 'has value',
dict2[eachKey]
...
dict2 key name has value earth
dict2 key port has value 80

```

```

>>> dict2= { 'host':'earth', 'port':80 }
>>> dict3= { 'host':'venus', 'server':'http' }
>>> dict2.update(dict3)
>>> dict2
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict3.clear()
>>> dict3
{}

>>> dict4 = dict2.copy()
>>> dict4
{'server': 'http', 'port': 80, 'host': 'venus'}
>>> dict4.get('host')
'venus'
>>> dict4.get('xxx')
>>> type(dict4.get('xxx'))
<type 'None'>
>>> dict4.get('xxx', 'no such key')
'no such key'

```

Conditionals and Loops

The primary focus of this chapter are Python's conditional and looping statements, and all their related components. We will take a close look at **if**, **while**, **for**, and their friends **else**, **elif**, **break**, **continue**, and **pass**.

Simple if statement

The **if** statement for Python will seem amazingly familiar; it is made up of three main components: the keyword itself, an expression which is tested for its truth value, and a code suite to execute if the expression evaluates to non-zero or true. The syntax for an **if** statement:

```
If expression:  
    expr_true_suite
```

The suite of the **if** clause, *expr_true_suite*, will be executed only if the above conditional expression results in a Boolean true value. Otherwise, execution resumes at the next statement following the suite.

```
a = int(raw_input("Enter the A Value"))  
b = int(raw_input("Enter the B Value"))
```

```
if(a>b):  
    print "A is Big"
```

if....else Statements

```
a = int(raw_input("Enter the A Value"))  
b = int(raw_input("Enter the B Value"))
```

```
if(a>b):  
    print "A is Big"  
else:  
    print "B is Big"
```

Nested if....else Statements

```
a = int(raw_input("Enter the A Value"))  
b = int(raw_input("Enter the B Value"))  
c = int(raw_input("Enter the C Value"))
```

```
if(a>b):  
    if(a>c):  
        print "A is Big"  
    else:  
        print "C is Big"  
else:  
    if(b>c):  
        print "B is Big"  
    else:  
        print "C is Big"
```

Multiple Condition in if....else Statements

```
a = int(raw_input("Enter the A Value"))
b = int(raw_input("Enter the B Value"))
c = int(raw_input("Enter the C Value"))
```

```
if(a>b and a>c):
    print "A is Big"
else:
    if(b>c):
        print "B is Big"
    else:
        print "C is Big"
```

elif Statements

```
mark1 = int(raw_input("Enter the Mark1"))
mark2 = int(raw_input("Enter the Mark2"))
mark3 = int(raw_input("Enter the Mark3"))

total = mark1 + mark2 + mark3

average = total / 3
```

```
if(mark1>=50 and mark2>=50 and
mark3>=50):
```

```
    result = "Pass"
```

```
else:
```

```
    result = "Fail"
```

```
if(mark1>=50 and mark2>=50 and
mark3>=50):
```

```
    if(average>=80):
```

```
        class1 = "First Class with Distinction"
```

```
    elif(average>=60 and average<80):
```

```
        class1 = "First Class"
```

```
    elif(average>=50 and average<60):
```

```
        class1 = "Second Class"
```

```
else:
```

```
    class1 = "No Class"
```

```
print "The Mark1 is:",mark1
print "The Mark2 is:",mark2
print "The Mark3 is:",mark3
print "The Total Value is:",total
print "The Average Value is:",average
print "The Student Result is:",result    80
print "The Student Class is:",class1
```


while Statement

Python's **while** is the first looping statement we will look at in this chapter. In fact, it is a conditional looping statement. In comparison with an **if** statement where a true expression will result in a single execution of the **if** clause suite, the suite in a **while** clause will be executed continuously in a loop until that condition is no longer satisfied.

General Syntax

```
while expression:  
    suite_to_repeat
```

The *suite_to_repeat* clause of the **while** loop will be executed continuously in a loop until expression evaluates to Boolean false. This type of looping mechanism is often used in a counting situation, such as the example in the next subsection.

```
n = int(raw_input("Enter the Number: "))  
i = int(raw_input("Enter the Initiale Value: "))  
  
while(i<=n):  
    print "The I Value is:",i  
    i += 1
```

for Statement

The other looping mechanism in Python comes to us in the form of the **for** statement. Unlike the traditional conditional looping **for** statement found in mainstream thirdgeneration languages (3GLs) like C, Fortran, or Pascal, Python's **for** is more akin to a scripting language's iterative foreach loop.

General Syntax

Iterative loops index through individual elements of a set and terminate when all the items are exhausted. Python's **for** statement iterates only through sequences, as indicated in the general syntax here:

```
for iter_var in sequence:  
    suite_to_repeat
```

The sequence *sequence* will be iterated over, and with each loop, the *iter_var* iteration variable is set to the current element of the sequence, presumably for use in *suite_to_repeat*.

Used with Sequence Types

In this section, we will see how the **for** loop works with the different sequence types. The examples will include string, list, and tuple types.

```
>>> for eachLetter in 'Names':  
    print 'current letter:', eachLetter
```

```
current letter: N  
current letter: a  
current letter: m  
current letter: e  
current letter: s
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself:

```
>>> nameList = ['Shirley', 'Terry', 'Joe', 'Heather', 'Lucy']  
>>> for nameIndex in range(len(nameList)):  
    print "Liu,", nameList[nameIndex]
```

```
Liu, Shirley  
Liu, Terry  
Liu, Joe  
Liu, Heather  
Liu, Lucy
```

range() Full Syntax

Python presents two different ways to use range(). The full syntax requires that two or all three integer arguments are present:

```
range( start, end, step=1)
```

range() will then return a list where for any k , $\text{start} \leq k < \text{end}$ and k iterates from start to end in increments of step. step cannot be 0, or else an error condition will occur.

```
>>> range(2, 19, 3)
```

```
[2, 5, 8, 11, 14, 17]
```

If step is omitted and only two arguments given, step takes a default value of 1.

```
>>> range(3,7)
```

```
[3, 4, 5, 6]
```

Let's take a look at an example used in the interpreter environment:

```
>>> for eachVal in range(2, 19, 3):
```

```
    print "value is:", eachVal
```

```
value is: 2
```

```
value is: 5
```

```
value is: 8
```

```
value is: 11
```

```
value is: 14
```

```
value is: 17
```

range() Abbreviated Syntax

range() also has a simple format, which takes one or both integer arguments:

```
range( start=0, end)
```

Given both values, this shortened version of range() is exactly the same as the long version of range() taking two parameters with step defaulting to 1. However, if given only a single value, start defaults to zero, and range() returns a list of numbers from zero up to the argument end:

```
>>> range(5)
[0, 1, 2, 3, 4]
```

We will now take this to the Python interpreter and plug in **for** and **print** statements to arrive at:

```
>>> for count in range(5):
    print count
...
0
1
2
3
4
```

Once **range()** executes and produces its list result, our expression above is equivalent to the following:

```
>>> for count in [0, 1, 2, 3, 4]:
    print count
```

NOTE

Now that you know both syntaxes for **range()**, one nagging question you may have is, why not just combine the two into a single one that looks like this?

```
range( start=0, end, step=1)# invalid
```

This syntax will work for a single argument or all three, but not two. It is illegal because the presence of **step** requires start to be given. In other words, you cannot provide **end** and **step** in a two-argument version because they will be (mis)interpreted as **start** and **end**.

xrange() Function for Limited Memory Situations

xrange() is similar to **range()** except that if you have a really large range list, **xrange()** may come in more handy because it does not have to make a complete copy of the list in memory. This built-in was made for exclusive use in **for** loops. It doesn't make sense outside a **for** loop. Also, as you can imagine, the performance will not be as good because the entire list is *not* in memory. Now that we've covered all the loops Python has to offer, let us take a look at the peripheral commands that typically go together with loops. These include statements to abandon the loop (**break**) and to immediately begin the next iteration (**continue**).

break Statement

The **break** statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional **break** found in C. The most common use for **break** is when some external condition is triggered (usually by testing with an **if** statement), requiring a hasty exit from a loop. The **break** statement can be used in both **while** and **for** loops.

```
count = num / 2
while count > 0:
    if (num % count == 0):
        print count, 'is the largest factor of', num
        break
    count = count - 1
```

The task of this piece of code is to find the largest divisor of a given number `num`. We iterate through all possible numbers that could possibly be factors of `num`, using the `count` variable and decrementing for every value that does NOT divide `num`. The first number that evenly divides `num` is the largest factor, and once that number is found, we no longer need to continue and use **break** to terminate the loop.

continue Statement

Whether in Python, C, Java, or any other structured language which features the **continue** statement, there is a misconception among some beginning programmers that the traditional **continue** statement "immediately starts the next iteration of a loop." While this may seem to be the apparent action, we would like to clarify this somewhat invalid supposition. Rather than beginning the next iteration of the loop when a **continue** statement is encountered, a **continue** statement terminates or discards the remaining statements in the current loop iteration and goes back to the top.

If we are in a conditional loop, the conditional expression is checked for validity before beginning the next iteration of the loop. Once confirmed, then the next iteration begins. Likewise, if the loop were iterative, a determination must be made as to whether there are any more arguments to iterate over. Only when that validation has completed successfully can we begin the next iteration.

```
valid = 0
count = 3
while count > 0:
    input = raw_input("enter password")
    # check for valid passwd
    for eachPasswd in passwdList:
        if input == eachPasswd:
            valid = 1
            break
```

```
if not valid: # (or valid == 0)
    print "invalid input"
    count = count - 1
    continue
else:
    break
```


pass Statement

One Python statement not found in C is the **pass** statement. Because Python does not use curly braces to delimit blocks of code, there are places where code is syntactically required. We do not have the equivalent empty braces or single semicolon the way C has to indicate "do nothing." If you use a Python statement that expects a sub-block of code or suite, and one is not present, you will get a syntax error condition. For this reason, we have **pass**, a statement that does absolutely nothing—it is a true NOP, to steal the "No OPeration" assembly code jargon. Style- and development-wise, **pass** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

```
def foo_func():  
    pass
```

or

```
if user_choice == 'do_calc':  
    pass  
else:  
    pass
```

FUNCTIONS

What Are Functions?

1. Functions are the structured or procedural programming way of organizing the logic in your programs.
2. Large blocks of code can be neatly segregated into manageable chunks, and space is saved by putting oft-repeated code in functions as opposed to multiple copies everywhere—this also helps with consistency because changing the single copy means you do not have to hunt for and make changes to multiple copies of duplicated code.
3. The basics of functions in Python are not much different from those of other languages with which you may be familiar.

Functions can appear in different ways... here is a sampling profile of how you will see functions created, used, or otherwise referenced:

<i>declaration/definition</i>	<code>def foo(): print 'bar'</code>
<i>function object/reference</i>	<code>foo</code>
<i>function call/invoke</i>	<code>foo()</code>

Functions vs. Procedures

1. Functions are often compared to procedures. Both are entities which can be invoked, but the traditional function or "black box," perhaps taking some or no input parameters, performs some amount of processing and concludes by sending back a return value to the caller.
2. Some functions are Boolean in nature, returning a "yes" or "no" answer, or, more appropriately, a non-zero or zero value, respectively. Procedures, often compared to functions, are simply special cases, functions which do not return a value.
3. As you will see below, Python procedures are implied functions because the interpreter implicitly returns a default value of None.

Return Values and Function Types

1. Functions may return a value back to its caller and those which are more procedural in nature do not explicitly return anything at all.
2. Languages which treat procedures as functions usually have a special type or value name for functions that "return nothing."
3. These functions default to a return type of "void" in C, meaning no value returned.
4. In Python, the equivalent return object type is None.

The `hello()` function acts as a procedure in the code below, returning no value. If the return value is saved, you will see that its value is `None`:

```
>>> def hello():  
    print 'hello world'
```

```
>>> res = hello()  
hello world
```

```
>>> res  
>>> print res  
None
```

```
>>> type(res)  
<type 'None'>
```

Also, like most other languages, you may return only one value/object from a function in Python. One difference is that in returning a container type, it will seem as if you can actually return more than a single object. In other words, you can't leave the grocery store with multiple items, but you can throw them all in a single shopping bag which you walk out of the store with, perfectly legal.

```
def foo():  
    return ['xyz', 1000000, -98.6]
```

```
def bar():  
    return 'abc', [42, 'python', "Guido"]
```

The foo() function returns a list, and the bar() function returns a tuple. Because of the tuple's syntax of not requiring the enclosing parentheses, it creates the perfect illusion of returning multiple items. If we were to properly enclose the tuple items, the definition of bar() would look like:

```
def bar():  
    return ('abc', [4-2j, 'python'], "Guido")
```

As far as return values are concerned, tuples can be saved in a number of ways. The following three ways of saving the return values are equivalent:

```
>>> aTuple = bar()  
>>> x, y, z = bar()  
>>> (a, b, c) = bar()  
>>> aTuple  
(('abc', [(4-2j), 'python'], 'Guido'))  
>>> x, y, z  
(('abc', [(4-2j), 'python'], 'Guido'))  
>>> (a, b, c)  
(('abc', [(4-2j), 'python'], 'Guido'))
```

Many languages which support functions maintain the notion that a function's type is the type of its return value. In Python, no direct type correlation can be made since Python is dynamically-typed and functions can return values of different types. Because overloading is not a feature

Calling Functions

Function Operator

Functions are called using the same pair of parentheses that you are used to. In fact, some consider (`()`) to be a two-character operator, the function operator. As you are probably aware, any input parameters or arguments must be placed between these calling parentheses. Parentheses are also used as part of function declarations to define those arguments.

Keyword Arguments

The concept of keyword arguments applies only to function invocation. The idea here is for the caller to identify the arguments by parameter name in a function call. This specification allows for arguments to be missing or out-of-order because the interpreter is able to use the provided keywords to match values to parameters. For a simple example, imagine a function `foo()` which has the following pseudocode definition:

```
def foo(x):  
    foo_suite # presumably does so processing with 'x'
```

Standard calls to foo(): foo(42) foo('bar') foo(y)

Keyword calls to foo(): foo(x=42) foo(x='bar') foo(x=y)

For a more realistic example, let us assume you have a function called net_conn() and you know that it takes two parameters, say, host and port:

```
def net_conn(host, port):  
    net_conn_suite
```

Naturally, we can call the function giving the proper arguments in the correct positional order which they were declared:

```
net_conn('kappa', 8080)
```

The host parameter gets the string 'kappa' and port gets 8080. Keyword arguments allow out-of-order parameters, but you must provide the name of the parameter as a "keyword" to have your arguments match up to their corresponding argument names, as in the following:

```
net_conn(port=8080, host='chino')
```


Creating Functions

def Statement

Functions are created using the **def** statement, with a syntax like the following:

```
def function_name(arguments):  
    "function_documentation_string"  
    function_body_suite
```

The header line consists of the **def** keyword, the function name, and a set of arguments (if any). The remainder of the **def** clause consists of an optional but highly-recommended documentation string and the required function body suite. We have seen many function declarations throughout this text, and here is another:

```
def helloSomeone(who):  
    'returns a salutory string customized with the input'  
    return "Hello" + str(who)
```

Declaration vs. Definition

Some programming languages differentiate between function declarations and function definitions. A function declaration consists of providing the parser with the function name, and the names (and traditionally the types) of its arguments, without necessarily giving any lines of code for the function, which is usually referred to as the function definition. In languages where there is a distinction, it is usually because the function definition may belong in a physically different location in the code from the function declaration. Python does not make a distinction between the two, as a function clause is made up of a declarative header line which is immediately followed by its defining suite.

Forward References

Like some other high-level languages, Python does not permit you to reference or call a function before it has been declared. We can try a few examples to illustrate this:

```
def foo():  
    print 'in foo()'  
    bar()
```

If we were to call `foo()` here, it will fail because `bar()` has not been declared yet:

```
>>> foo()  
in foo()  
Traceback (innermost last):  
File "<stdin>", line 1, in ?  
File "<stdin>", line 3, in foo  
NameError: bar
```

We will now define `bar()`, placing its declaration before `foo()`'s declaration:

```
def bar():  
    print 'in bar()'  
  
def foo():  
    print 'in foo()'  
    bar()
```

Now we can safely call `foo()` with no problems:

```
>>> foo()  
in foo()  
in bar()
```

In fact, we can even declare `foo()` before `bar()`:

```
def foo():  
    print 'in foo()'  
    bar()  
  
def bar():  
    print 'in bar()'
```

Amazingly enough, this code still works fine with no forward referencing problems:

```
>>> foo()  
in foo()  
in bar()
```

This piece of code is fine because even though a call to `bar()` (from `foo()`) appears before `bar()`'s definition, `foo()` *itself* is not called before `bar()` is declared. In other words, we declared `foo()`, then `bar()`, and *then* called `foo()`, but by that time, `bar()` existed already, so the call succeeds. Notice that the output of `foo()` succeeded before the error came about. `NameError` is the exception that is always raised when any uninitialized identifiers are accessed.

Passing Functions

The concept of function pointers is an advanced topic when learning a language such as C, but not Python where functions are like any other object. They can be referenced (accessed or aliased to other variables), passed as arguments to functions, be elements of container objects like lists and dictionaries, etc. The one unique characteristic of functions which may set them apart from other objects is that they are callable, i.e., can be invoked via the function operator. Because all objects are passed by reference, functions are no different. When assigning to another variable, you are assigning the reference to the same object; and if that object is a function, then all aliases to that same object are invokable:

```
>>> def foo():  
...     print 'in foo()'  
...  
>>> bar = foo  
>>> bar()  
in foo()
```

When we assigned `foo` to `bar`, we are assigning the same function object to `bar`, thus we can invoke `bar()` in the same way we call `foo()`. Be sure you understand the difference between `"foo"` (reference of the function object) and `"foo()"` (invocation of the function object) Taking our reference example a bit further, we can even pass functions in as arguments to other functions for invocation:

```
>>> def bar(argfunc):  
...   argfunc()  
...  
>>> bar(foo)  
in foo()
```

Note that it is the function object `foo` that is being passed to `bar()`. `bar()` is the function that actually calls `foo()` (which has been aliased to the local variable `argfunc` in the same way that we assigned `foo` to `bar`).

```
def convert(func, seq):
    newSeq = [ ]
    for eachNum in seq:
        newSeq.append(func(eachNum))
    return newSeq

def test():
    myseq = (123, 45.67, -6.2e8, 999999999L)
    print convert(int, myseq)
    print convert(long, myseq)
    print convert(float, myseq)

test()
```

Output

```
[123, 45, -620000000, 999999999]
```

```
[123L, 45L, -620000000L, 999999999L]
```

```
[123.0, 45.67, -620000000.0, 999999999.0]
```

Formal Arguments

A Python function's set of formal arguments consists of all parameters passed to the function on invocation for which there is an exact correspondence to those of the argument list in the function declaration. These arguments include all required arguments (passed to the function in correct positional order), keyword arguments (passed in- or outof- order, but which have keywords present to match their values to their proper positions in the argument list), and all arguments which have default values which may or may not be part of the function call. For all of these cases, a name is created for that value in the (newly-created) local namespace and can be accessed as soon as the function begins execution.

```
def addition(x,y): # Formal Aruguments
    a = x
    b = y
    c = a + b
    print "Addition of the Two Numbers:",c

a1 = int(raw_input("Enter the a1 Value"))
b1 = int(raw_input("Enter the a2 Value"))

addition(a1,b1)    #Actual Arguments
```

Positional Arguments

These are the standard vanilla parameters that we are all familiar with. Positional arguments must be passed in the exact order that they are defined for the functions that are called. Also, without the presence of any default arguments (see next section), the exact number of arguments passed to a function (call) must be exactly the number declared:

```
>>> def foo(who): # defined for only 1 argument
...     print 'Hello', who
...
>>> foo() # 0 arguments... BAD
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: not enough arguments; expected 1, got 0
>>>
>>> foo('World!') # 1 argument... WORKS
Hello World!
>>>
>>> foo('Mr.', 'World!') # 2 arguments... BAD
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: too many arguments; expected 1, got 2
```


Default Arguments

Default arguments are parameters which are defined to have a default value if one is not provided in the function call for that argument. Such definitions are given in the function declaration header line. C++ and Java are other languages which support default arguments and whose declaration syntax is shared with Python: The argument name is followed by an “assignment of its default value. This assignment is merely a syntactical way of indicating that this assignment will occur if no value is passed in for that argument.

The syntax for declaring variables with default values in Python is such that all positional arguments must come before any default arguments:

```
def function_name(posargs, defarg1=dval1, defarg2=dval2,...):  
    "function_documentation_string"  
    function_body_suite
```

Each default argument is followed by an assignment statement of its default value. If no value is given during a function call, then this assignment is realized.

Why Default Arguments?

1. Default arguments add a wonderful level of robustness to applications because they allow for some flexibility that is not offered by the standard positional parameters.
2. That gift comes in the form of simplicity for the applications programmer. Life is not as complicated when there are a fewer number of parameters that one needs to worry about.
3. This is especially helpful when one is new to an API interface and does not have enough knowledge to provide more targeted values as arguments.
4. The concept of using default arguments is analogous to the process of installing software on your computer.
5. How often does one chose the "default install" over the "custom install?" I would say probably almost always. It is a matter of convenience and knowhow, not to mention a timesaver.
6. And if you *are* one of those gurus who always chooses the custom install, please keep in mind that you are one of the minority.

```
def taxMe(cost, rate=0.0825):  
    return cost + (cost * rate)
```

```
print taxMe(100)  
print taxMe(100, 0.05)
```

Variable-length Arguments

There may be situations where your function is required to process an unknown number of arguments. These are called *variable-length argument lists*. Variable-length arguments are not named explicitly in function declarations because the number of arguments is unknown before runtime (and even during execution, the number of arguments may be different on successive calls), an obvious difference from formal arguments (positional and default) which *are* named in function declarations. Python supports variable-length arguments in two ways because function calls provide for both keyword and non-keyword argument types.

EXCEPTION IN PYTHON

As you were going through some of the examples in the previous chapters, you no doubt noticed what happens when your program "crashes" or terminates due to unresolved errors. A "traceback" notice appears along with a notice with as much diagnostic information as the interpreter can give you, including the error name, reason, and perhaps even the line number near or exactly where the error occurred. All errors have a similar format, regardless of whether running within the Python interpreter or standard script execution, providing a consistent error interface. All errors, whether they be syntactical or logical, result from behavior incompatible with the Python interpreter and cause exceptions to be raised.

NameError: attempt to access an undeclared variable >>> foo Traceback (innermost last): File "<interactive input>", line 0, in ? NameError: foo	ZeroDivisionError: division by any numeric zero >>> 12.4/0.0 Traceback (innermost last): File "<interactive input>", line 0, in ? ZeroDivisionError: float division
IndexError: request for an out-of-range index for sequence >>> aList = [] >>> aList[0] Traceback (innermost last): File "<stdin>", line 1, in ? IndexError: list index out of range	KeyError: request for a non-existent dictionary key >>> aDict = {'host': 'earth', 'port': 80} >>> print aDict['server'] Traceback (innermost last): File "<stdin>", line 1, in ? KeyError: server

IOError: input/output error

```
>>> f = open("blah")
Traceback (innermost last):
File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory:
'blah'
```

AttributeError: attempt to access an unknown object attribute

```
>>> class myClass:
...     pass
...
>>> myInst = myClass()
>>> myInst.bar = 'spam'
>>> myInst.bar
'spam'
>>> myInst.foo
Traceback (innermost last):
File "<stdin>", line 1, in ?
AttributeError: foo
```

Detecting and Handling Exceptions

1. Exceptions can be detected by incorporating them as part of a **try** statement.
2. Any code suite of a **try** statement will be monitored for exceptions.
3. There are two main forms of the **try** statement: **try-except** and **try-finally**.
4. These statements are mutually exclusive, meaning that you pick only one of them.
5. A **try** statement is either accompanied by one or more **except** clauses or exactly one **finally** clause. (There is no such thing as a hybrid "try-except-finally.") **try-except** statements allow one to detect and handle exceptions.
6. There is even an optional **else** clause for situations where code needs to run only when no exceptions are detected.
7. Meanwhile, **try-finally** statements allow only for detection and processing of any obligatory clean-up (whether or not exceptions occur), but otherwise has no facility in dealing with exceptions.

try-except Statement

The **try-except** statement (and more complicated versions of this statement) allows you to define a section of code to monitor for exceptions and also provides the mechanism to execute handlers for exceptions.

The syntax for the most general **try-except** statement looks like this:

try:

try_suite # watch for exceptions here

except *Exception:*

except_suite # exception-handling code

Let us give one example, then explain how things work. We will use our `IOError` example from above. We can make our code more robust by adding a **try-except** "wrapper" around the code:

```
>>> try:
...     f = open('blah')
... except IOError:
...     print 'could not open file'
...
could not open file
```

As you can see, our code now runs seemingly without errors. In actuality, the same `IOError` still occurred when we attempted to open the nonexistent file. The difference? We added code to both detect and handle the error. When the `IOError` exception was raised, all we told the interpreter to do was to output a diagnostic message. The program continues and does not "bomb out" as our earlier example

Wrapping a Built-in Function

The `float()` built-in function has a primary purpose of converting any numeric type to a float. In Python 1.5, `float()` was given the added feature of being able to convert a number given in string representation to an actual float value, obsoleting the use of the `atof()` function of the `string` module. Readers with older versions of Python may still use `string.atof()`, replacing `float()`, in the examples we use here.


```
>>> float(12345)
12345.0
>>> float('12345')
12345.0
>>> float('123.45e67')
1.2345e+069
```

Unfortunately, `float()` is not very forgiving when it comes to bad input:

```
>>> float('abcde')
Traceback (innermost last):
File "<stdin>", line 1, in ?
float('abcde')
ValueError: invalid literal for float(): abcde
>>>
```

```
>>> float(['this is', 1, 'list'])
Traceback (innermost last):
File "<stdin>", line 1, in ?
float(['this is', 1, 'list'])
TypeError: object can't be converted to float
```

Notice in the errors above that `float()` does not take too kindly to strings which do not represent numbers or non-strings. Specifically, if the correct argument type was given (string type) but that type contained an invalid value, the exception raised would be `ValueError` because it was the value that was improper, not the type. In contrast, a list is a bad argument altogether, not even being of the correct type; hence, `TypeError` was thrown.

Our exercise is to call `float()` "safely," or in a more "safe manner," meaning that we want to ignore error situations because they do not apply to our task of converting numeric string values to floating point numbers, yet are not severe enough errors that we feel the interpreter should abandon execution. To accomplish this, we will create a "wrapper" function, and, with the help of **`try-except`**, create the environment that we envisioned. We shall call it `safe_float()`. In our first iteration, we will scan and ignore only `ValueErrors`, because they are the more likely culprit. `TypeError`s rarely happen since somehow a non-string must be given to `float()`.

```
def safe_float(object):  
    try:  
        return float(object)  
    except ValueError:  
        pass
```

The first step we take is to just "stop the bleeding." In this case, we make the error go away by just "swallowing it." In other words, the error will be detected, but since we have nothing in the **except** suite (except the **pass** statement, which does nothing but serve as a syntactical placeholder for where code is supposed to go), no handling takes place. We just ignore the error.

One obvious problem with this solution is that we did not explicitly return anything to the function caller in the error situation. Even though `None` is returned (when a function does not return any value explicitly, i.e., completing execution without encountering a **return** *object* statement), we give little or no hint that anything wrong took place. The very least we should do is to explicitly return `None` so that our function returns a value in both cases and makes our code somewhat easier to understand:

```
def safe_float(object):  
    try:  
        retval = float(object)  
    except ValueError:  
        retval = None  
    return retval
```

Bear in mind that with our change above, nothing about our code changed except that we used one more local variable. In designing a well-written application programmer interface (API), you may have kept the return value more flexible.

Perhaps you documented that if a proper argument was passed to `safe_float()`, then indeed, a floating point number would be returned, but in the case of an error, you chose to return a string indicating the problem with the input value. We modify our code one more time to reflect this change:

```
def safe_float(object):  
    try:  
        retval = float(object)  
    except ValueError:  
        retval = 'could not convert non-number to float'  
    return retval
```

The only thing we changed in the example was to return an error string as opposed to just `None`. We should take our function out for a "test drive" to see how well it works so far:

```
>>> safe_float('12.34')  
12.34
```

```
>>> safe_float('bad input')  
'could not convert non-number to float'
```

We made a good start—now we can detect invalid string input, but we are still vulnerable to invalid *objects* being passed in:

```
>>> safe_float({'a': 'Dict'})
Traceback (innermost last):
File "<stdin>", line 1, in ?
File "safeflt.py", line 28, in safe_float
  retval = float(object)
TypeError: object can't be converted to float
```

We will address this final shortcoming momentarily, but before we further modify our example, we would like to highlight the flexibility of the **try-except** syntax, especially the **except** statement, which comes in a few more flavors.

try Statement with Multiple excepts

The **except** statement in such formats specifically detects exceptions named *Exception*. You can chain multiple **except** statements together to handle different types of exceptions with the same

```
try:
except Exception1:
    suite_for_exception_Exception1
except Exception2:
    suite_for_exception_Exception2
:
```

This same **try** clause is attempted, and if there is no error, execution continues, passing all the **except** clauses. However, if an exception *does* occur, the interpreter will look through your list of handlers attempting to match the exception with one of your handlers (**except** clauses). If one is found, execution proceeds to *that* **except** suite.

Our `safe_float()` function has some brains now to detect specific exceptions. Even smarter code would handle each appropriately. To do that, we have to have separate **except** statements, one for each exception type. That is no problem as Python allows **except** statements can be chained together. Any reader familiar with popular thirdgeneration languages (3GLs) will no doubt notice the similarities to the switch/case statement which is absent in Python. We will now create separate messages for each error type, providing even more detail to the user as to the cause of his or her problem:

```
def safe_float(object):  
    try:  
        retval = float(object)  
    except ValueError:  
        retval = 'could not convert non-number to float'  
    except TypeError:  
        retval = 'object type cannot be converted to float'  
    return retval
```

Running the code above with erroneous input, we get the following:

```
>>> safe_float('xyz')  
'could not convert non-number to float'
```

```
>>> safe_float()  
'argument must be a string'
```

```
>>> safe_float(200L)  
200.0
```

```
>>> safe_float(45.67000)  
45.67
```

except Statement with Multiple Exceptions

We can also use the same **except** clause to handle multiple exceptions. **Except** statements which process more than one exception require that the set of exceptions be contained in a tuple:

```
except (Exception1, Exception2):  
    suite_for_Exception1_and_Exception2
```

The above syntax example illustrates how two exceptions can be handled by the same code. In general, any number of exceptions can follow an **except** statement as long as they are all properly enclosed in a tuple:

```
except (Exception1[, Exception2[, ... ExceptionN...]]):  
    suite_for_exceptions_Exception1_to_ExceptionN
```

If for some reason, perhaps due to memory constraints or dictated as part of the design that all exceptions for our `safe_float()` function must be handled by the same code, we can now accommodate that requirement:

```
def safe_float(object):  
    try:  
        retval = float(object)  
    except (ValueError, TypeError):  
        retval = 'argument must be a number or numeric string'  
    return retval
```

Now there is only the single error string returned on erroneous input:

```
>>> safe_float('Spanish Inquisition')  
'argument must be a number or numeric string'  
>>> safe_float([])  
'argument must be a number or numeric string'  
>>> safe_float('1.6')  
1.6  
>>> safe_float(1.6)  
1.6  
>>> safe_float(932)  
932.0
```


try-except with No Exceptions Named

The final syntax for **try-except** we are going to present is one which does not specify an exception on the except header line:

try:

try_suite # watch for exceptions here

except:

except_suite # handles all exceptions

Although this code "catches the most exceptions," it does not promote good Python coding style. One of the chief reasons is that it does not take into account the potential root causes of problems which may generate exceptions. Rather than investigating and discovering what types of errors may occur and how they may be prevented from happening, this type of code "turns the blind eye," thereby ignoring the possible causes

NOTE

The **try-except** statement has been included in Python to provide a powerful mechanism for programmers to track down potential errors and to perhaps provide logic within the code to handle situations where it may not otherwise be possible, for example in C. The main idea is to minimize the number of errors and still maintain program correctness. As with all tools, they must be used properly.

One incorrect use of **try-except** is to serve as a giant bandage over large pieces of code. By that we mean putting large blocks, if not your entire source code, within a **try** and/or have a large generic **except** to "filter" any fatal errors by ignoring them:

```
# this is really bad code
```

```
try:
```

```
    large_block_of_code # bandage of large piece of code
```

```
except:
```

```
    Pass
```

```
    # blind eye ignoring all errors
```

Obviously, errors cannot be avoided, and the job of **try-except** is to provide a mechanism whereby an acceptable problem can be remedied or properly dealt with, and not be used as a filter. The construct above will hide many errors, but this type of usage promotes a poor engineering practice that we certainly cannot endorse.

"Exceptional Arguments"

No, the title of this section has nothing to do with having a major fight. Instead, we are referring to the fact that exception may have *arguments* are passed along to the exception handler when they are raised. When an exception is raised, parameters are generally provided as an additional aid for the exception handler. Although arguments to exceptions are optional, the standard built-in exceptions do provide at least one argument, an error string indicating the cause of the exception.

Exception parameters can be ignored in the handler, but the Python provides syntax for saving this value. To access any provided exception argument, you must reserve a variable to hold the argument. This argument is given on the **except** header line and follows the exception type you are handling. The different syntaxes for the **except** statement can be extended to the following:

single exception

except *Exception*,

Argument: suite_for_Exception_with_Argument

multiple exceptions **except** (*Exception1*, *Exception2*, ..., *ExceptionN*),

Argument: suite_for_Exception1_to_ExceptionN_with_Argument

The example below is when an invalid object is passed to the float() built-in function, resulting in a TypeError exception:

```
>>> try:
... float(['float() does not', 'like lists', 2])
... except TypeError, diag:# capture diagnostic info
... pass
...
>>> type(diag)
<type 'instance'>
>>>
>>> print diag
object can't be converted to float
```

The first thing we did was cause an exception to be raised from within the **try** statement. Then we passed cleanly through by ignoring but saving the error information. Calling the `type()` built-in function, we were able to confirm that our exception was indeed an instance. Finally, we displayed the error by calling `print` with our diagnostic exception argument.

To obtain more information regarding the exception, we can use the special `__class__` instance attribute which identifies which class an instance was instantiated from. Class objects also have attributes, such as a documentation string and a string name which further illuminate the error type:

```
>>> diag # exception instance object
<exceptions.TypeError instance at 8121378>
```

```
>>> diag.__class__ # exception class object
<class exceptions.TypeError at 80f6d50>
```

```
>>> diag.__class__.__doc__ # exception class documentation string 'Inappropriate
argument type.'
```

```
>>> diag.__class__.__name__ # exception class name
'TypeError'
```

In the following code snippet, we replace our single error string with the string representation of the exception argument.

```
def safe_float(object):  
    try:  
        retval = float(object)  
    except (ValueError, TypeError), diag:  
        retval = str(diag)  
    return retval
```

Upon running our new code, we obtain the following (different) messages when providing improper input to `safe_float()`, even if both exceptions are managed by the same handler:

```
>>> safe_float('xyz')  
'invalid literal for float(): xyz'  
>>> safe_float({})  
'object can't be converted to float'
```

else Clause

We have seen the **else** statement with other Python constructs such as conditionals and loops. With respect to **try-except** statements, its functionality is not that much different from anything else you have seen: The **else** clause executes if no exceptions were detected in the preceding **try** suite.

All code within the **try** suite must have completed successfully (i.e., concluded with no exceptions raised) before any code in the **else** suite begins execution. Here is a short example in Python pseudocode:

```
import 3rd_party_module
log = open('logfile.txt', 'w')
try:
    3rd_party_module.function()
except:
    log.write("*** caught exception in module\n")
else:
    log.write("*** no exceptions caught\n")
    log.close()
```

In the above example, we import an external module and test it for errors. A log file is used to determine whether there were defects in the third-party module code. Depending on whether an exception occurred during execution of the external function, we write differing messages to the log.

try-finally Statement

The **try-finally** statement differs from its **try-except** brethren in that it is not used to handle exceptions. Instead it is used to maintain consistent behavior regardless of whether or not exceptions occur. The **finally** suite executes regardless of an exception being triggered within the **try** suite.

try:

try_suite

finally:

finally_suite # executes regardless of exceptions

When an exception does occur within the **try** suite, execution jumps immediately to the **finally** suite. When all the code in the **finally** suite completes, the exception is reraised for handling at the next higher layer. Thus it is common to see a **try-finally** nested as part of a **try-except** suite.

```
x = 10
y = 0
```

```
try:
    z = x / y
    print "Division of the Two Numbers:",z
except ArithmeticError:
    print "Arithmetic Error: Division By Zero"
finally:
    print "Can't identify the such error"
```

I / O:

```
>>>
```

```
Arithmetic Error: Division By Zero
Can't identify the such error
```

```
>>>
```

```
x = 10
y = 0
```

```
try:
```

```
    z = x / y
```

```
    print "Division of the Two Numbers:",z
```

```
finally:
```

```
    print "Can't identify the such error"
```

```
>>>
```

```
Can't identify the such error
```

Traceback (most recent call last):

File "C:/Python27/ee", line 5, in <module>

```
    z = x / y
```

```
ZeroDivisionError: integer division or
modulo by zero
```

```
>>>
```

Exceptions as Strings

```
# this may not work... risky!
```

```
try:
```

```
:
```

```
raise 'myexception'
```

```
:
```

```
except 'myexception':
```

```
    suite_to_handle_my_string_exception
```

```
except:
```

```
    suite_for_other_exceptions
```

```
# this is a little bit better
```

```
myexception = 'myexception'
```

```
try:
```

```
:
```

```
raise myexception
```

```
:
```

```
except myexception:
```

```
    suite_to_handle_my_string_exception
```

```
except:
```

```
    suite_for_other_exceptions
```

With this update, the same string object is used. However, if you are going to use this code, you might as well use an exception class. Substitute the myexception assignment above with:

this is the best choice

```
class MyException(Exception):  
    pass  
    :  
    try:  
    :  
        raise MyException  
    :  
    except MyException:  
        suite_to_handle_my_string_exception  
    except:  
        suite_for_other_exceptions
```

So you see, there really is no reason *not* to use exception classes from now on when creating your own exceptions. Be careful, however, because you may end up using an external module which may still have exceptions implemented as strings.

Raising Exceptions

The interpreter was responsible for raising all of the exceptions which we have seen so far. These exist as a result of encountering an error during execution. A programmer writing an API may also wish to throw an exception on erroneous input, for example, so Python provides a mechanism for the programmer to explicitly generate an exception: the **raise** statement.

raise Statement

The **raise** statement is quite flexible with the arguments which it supports, translating to a large number of different formats supported syntactically. The general syntax for **raise** is:

```
raise [Exception [, args [, traceback]]]
```

The first argument, *Exception*, is the name of the exception to raise. If present, it must either be a string, class, or instance (more below). *Exception* must be given if any of the other arguments (arguments or *traceback*) are present. A list of all Python standard exceptions is given in Table 10.2. The second expression contains optional *args* (a.k.a. parameters, values) for the exception. This value is either a single object or a tuple of objects. When exceptions are detected, the exception arguments are always returned as a tuple. If *args* is a tuple, then that tuple represents the same set of exception arguments which are given to the handler.

If *args* is a single object, then the tuple will consist solely of this one object (i.e., a tuple with one element). In most cases, the single argument consists of a string indicating the cause of the error. When a tuple is given, it usually equates to an error string, an error number, and perhaps an error location, such as a file, etc.

The final argument, *traceback*, is also optional (and rarely used in practice), and, if present, is the traceback object used for the exception—normally a traceback object is newly created when an exception is raised. This third argument is useful if you want to re-raise an exception (perhaps to point to the previous location from the current). Arguments which are absent are represented by the value **None**.

The most common syntax used is when *Exception* is a class. No additional parameters are ever required, but in this case, if they are given, can be a single object argument, a tuple of arguments, or an exception class instance. If the argument is an instance, then it can be an instance of the given class or a derived class (subclassed from a pre-existing exception class). No additional arguments (i.e., exception arguments) are permitted if the argument is an instance.

What happens if the argument is an instance? No problems arise if instance is an instance of the given exception class. However, if instance is *not* an instance of the class nor an instance of a subclass of the class, then a new instance of the exception class will be created with exception arguments copied from the given instance. If instance is an instance of a subclass of the exception class, then the new exception will be instantiated from the subclass, not the original exception class.

If the additional parameter to the **raise** statement used with an exception class is not an instance—instead, it is a singleton or tuple—then the class is instantiated and *args* is used as the argument list to the exception. If the second parameter is not present or **None**, then the argument list is empty.

to illuminate all the different ways which **raise** can be used.

Table 10.1. Using the raise Statement	
raise <i>syntax</i>	<i>Description</i>
raise <i>exclass</i>	raise an exception, creating an instance of <i>exclass</i> (without any exception arguments)
raise <i>exclass</i> ()	same as above since classes are now exceptions; invoking the class name with the function call operator instantiates an instance of <i>exclass</i> , also with no arguments
raise <i>exclass</i> , <i>args</i>	same as above, but also providing exception arguments <i>args</i> , which can be a single argument or a tuple
raise <i>exclass</i> (<i>args</i>)	same as above
raise <i>exclass</i> , <i>args</i> , <i>tb</i>	same as above, but provides traceback object <i>tb</i> to use
raise <i>exclass</i> , <i>instance</i>	raise exception using <i>instance</i> (normally an instance of <i>exclass</i>); if <i>instance</i> is an instance of a subclass of <i>exclass</i> , then the new exception will be of the subclass type (not of <i>exclass</i> type); if <i>instance</i> is <i>not</i> an instance of <i>exclass</i> <i>nor</i> an instance of a subclass of <i>exclass</i> , then a new instance of <i>exclass</i> will be created with exception arguments copied from <i>instance</i>
raise <i>instance</i>	raise exception using <i>instance</i> : the exception type is the class which instantiated <i>instance</i> ; equivalent to raise

Assertions

Assertions are diagnostic predicates which must evaluate to Boolean true; otherwise, an exception is raised to indicate that the expression is false. These work similarly to the `assert` macros which are part of the C language preprocessor, but in Python these are runtime constructs (as opposed to pre-compile directives).

If you are new to the concept of assertions, no problem. The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a **raise-if-not** statement). An expression is tested, and if the result comes up false, an exception is raised.

assert Statement

The **assert** statement evaluates a Python expression, taking no action if the assertion succeeds (similar to a **pass** statement), but otherwise raises an `AssertionError` exception. The syntax for **assert** is:

```
assert expression[, arguments]
```

Here are some examples of the use of the **assert** statement:

```
assert 1 == 1
assert (2 + 2) == (2 * 2)
assert len(['my list', 12]) < 10
assert range(3) == [0, 1, 2]
```

AssertionError exceptions can be caught and handled like any other exception using the **try-except** statement, but if not handled, they will terminate the program and produce a traceback similar to the following:

```
>>> assert 1 == 0
Traceback (innermost last):
File "<stdin>", line 1, in ?
AssertionError
```

Like the **raise** statement we investigated in the previous section, we can provide an exception argument to our **assert** command:

```
>>> assert 1 == 0, 'One does not equal zero silly!'
Traceback (innermost last):
File "<stdin>", line 1, in ?
AssertionError: One does not equal zero silly!
```

Here is how we would use a **try-except** statement to catch an AssertionError exception:

```
try:
    assert 1 == 0, 'One does not equal zero silly!'
except AssertionError, args:
    print '%s: %s' % (args.__class__.__name__, args)
```

Executing the above code from the command-line would result in the following output:

```
AssertionError: One does not equal zero silly!
```

Standard Exceptions

Table 10.2. Python Standard Exceptions

<i>Exception Name</i>	<i>Description</i>
<code>Exception</code> ^[a]	root class for all exceptions
<code>SystemExit</code>	request termination of Python interpreter
<code>StandardError</code> ^[a]	base class for all standard built-in exceptions
<code>ArithmeticError</code> ^[a]	base class for all numeric calculation errors
<code>FloatingPointError</code> ^[a]	error in floating point calculation
<code>OverflowError</code>	calculation exceeded maximum limit for numerical type
<code>ZeroDivisionError</code>	division (or modulus) by zero error (all numeric types)
<code>AssertionError</code> ^[a]	failure of <code>assert</code> statement
<code>AttributeError</code>	no such object attribute
<code>EOFError</code>	end-of-file marker reached without input from built-in
<code>EnvironmentError</code> ^[b]	base class for operating system environment errors
<code>IOError</code>	failure of input/output operation
<code>OSError</code> ^[b]	operating system error
<code>WindowsError</code> ^[c]	MS Windows system call failure
<code>ImportError</code>	failure to import module or object
<code>KeyboardInterrupt</code>	user interrupted execution (usually by typing ^C)
<code>LookupError</code> ^[a]	base class for invalid data lookup errors
<code>IndexError</code>	no such index in sequence

KeyError	no such key in mapping
MemoryError	out-of-memory error (non-fatal to Python interpreter)
NameError	undeclared/uninitialized object (non-attribute)
UnboundLocalError ^[c]	access of an uninitialized local variable
RuntimeError	generic default error during execution
NotImplementedError ^[b]	unimplemented method
SyntaxError	error in Python syntax
IndentationError ^[d]	improper indentation
TabError ^[d]	improper mixture of TABs and spaces
SystemError	generic interpreter system error
TypeError	invalid operation for type

ValueError

```
try:  
    float('xyz')  
except ValueError:  
    print "The String is Cannot convert Float"
```

TypeError

```
try:  
    a = ['xyz',10.0,68]  
    float(a)  
except TypeError:  
    print "The Object type is Cannot Convert Float Type"
```

ArithmeticError

```
try:  
    a = 10  
    b = 0  
    c = a / b  
    print c  
except ArithmeticError:  
    print "Division By Error"
```

ZeroDivisionError

try:

```
a = 10
```

```
b = 0
```

```
c = a / b
```

```
print c
```

```
except ZeroDivisionError:
```

```
    print "Division By Error"
```

Exception

try:

```
    print a
```

```
except Exception:
```

```
    print "The Variable is Not defined"
```

OverflowError

```
import math
```

try:

```
    math.exp(1000) / math.exp(1000)
```

```
except OverflowError:
```

```
    print "The Overflow Error"
```

KeyError

try:

```
    a = {10:100,20:200}
```

```
    print a
```

```
    print a[50]
```

```
except KeyError:
```

```
    print "The Key Value is not defined in the  
Dictionary"
```

NameError

try:

```
    print a
```

```
except NameError:
```

```
    print "a is Not Defined"
```

IndexError

try:

```
    a = [10,20,30,40]
```

```
    print a
```

```
    print a[9]
```

```
except IndexError:
```

```
    print "The Location is Not Identified"
```

IOError

```
try:  
    f = open("ccet.txt")  
except IOError:  
    print "The File is Cannot be Opened"
```

AttributeError

```
class Example:  
    pass  
try:  
    e = Example()  
    e.a = 1234  
    print e.a  
    print e.b  
except AttributeError:  
    print "The Instance of the Class is Not Defined"
```

StandardError

```
try:  
    a = [10,20,30,40]  
    print a  
    print a[9]  
except StandardError:  
    print "The Location is Not Identified"
```

Files

and

Input / Output

File Objects

File objects can be used not only to access normal disk files, but also any other type of "file" that uses that abstraction. Once the proper "hooks" are installed, you can access other objects with file-like interfaces in the same manner you would access normal files. The `open()` built-in function (see below) returns a file object which is then used for all succeeding operations on the file in question. There are a large number of other functions which return a file or file-like object. One primary reason for this abstraction is that many input/output data structures prefer to adhere to a common interface. It provides consistency in behavior as well as implementation. Operating systems like Unix even feature files as an underlying and architectural interface for communication. Remember, files are simply a contiguous sequence of bytes. Anywhere data needs to be sent usually involves a byte stream of some sort, whether the stream occurs as individual bytes or blocks of data.

File Built-in Function [`open()`]

As the key to opening file doors, the `open()` built-in function provides a general interface to initiate the file input/output (I/O) process. `open()` returns a file object on a successful opening of the file or else results in an error situation. The basic syntax of the `open()` built-in function is:

`file_object = open(file_name, access_mode='r', buffering=-1)`¹⁴¹

The *file_name* is a string containing the name of the file to open. It can be a relative or absolute/full pathname. The *access_mode* optional variable is also a string, consisting of a set of flags indicating which mode to open the file with. Generally, files are opened with the modes "r," "w," or "a," representing read, write, and append, respectively. Any file opened with mode "r" must exist. Any file opened with "w" will be truncated first if it exists, and then the file is (re)created. Any file opened with "a" will be opened for write. If the file exists, the initial position for file (write) access is set to the end-of-file. If the file does not exist, it will be created, making it the same as if you opened the file in "w" mode. If you are a C programmer, these are the same file open modes used for the C library function `fopen()`.

There are other modes supported by `fopen()` that will work with Python's `open()`. These include the "+" for read-write access and "b" for binary access.

The other optional argument, *buffering*, is used to indicate the type of buffering that should be performed when accessing the file. A value of 0 means no buffering should occur, a value of 1 signals line buffering, and any value greater than 1 indicates buffered I/O with the given value as the buffer size. The lack of or a negative value indicates that the system default buffering scheme should be used, which is line buffering for any teletype or tty-like device and normal buffering for everything else. Under normal circumstances, a *buffering* value is not given, thus using the system default.

Table 9.1. Access Modes for File Objects

<i>File Mode</i>	<i>Operation</i>
r	open for read
w	open for write (truncate if necessary)
a	open for write (start at EOF, create if necessary)
r+	open for read and write
w+	open for read and write (see "w" above)
a+	open for read and write (see "a" above)
rb	open for binary read
wb	open for binary write (see "w" above)
ab	open for binary append (see "a" above)
rb+	open for binary read and write (see "r+" above)
wb+	open for binary read and write (see "w+" above)
ab+	open for binary read and write (see "a+" above)

Here are some examples for opening files:

```
fp = open('/etc/motd')      #open file for read
fp = open('test', 'w')     #open file for write
fp = open('data', 'r+')    #open file for read/write
fp = open('c:\io.sys', 'rb') #open binary file for read
```

File Built-in Methods

Once `open()` has completed successfully and returned a file object, all subsequent access to the file transpires with that "handle." File methods come in four different categories:

1. input
2. output
3. movement within a file, which we will call "intra-file motion"
4. miscellaneous.

Input

The `read()` method is used to read bytes directly into a string, reading at most the number of bytes indicated. If no size is given, the default value is set to `-1`, meaning that the file is read to the end. The `readline()` method reads one line of the open file (reads all bytes until a `NEWLINE` character is encountered). The `NEWLINE` character is retained in the returned string. The `readlines()` method is similar, but reads all remaining lines as strings and returns a list containing the read set of lines.

Output

The `write()` built-in method has the opposite functionality as `read()` and `readline()`. It takes a string which can consist of one or more lines of text data or a block of bytes and writes the data to the file. `writelines()` operates on a list just like `readlines()`, but takes a list of strings and writes them out to a file. NEWLINE characters are not inserted between each line; so if desired, they must be added to the end of each line before `writelines()` is called.

```
>>> output=['1stline', '2ndline', 'the end']
>>> [x + '\n' for x in output]
['1stline\n', '2ndline\n', 'the end\n']
```

Note that there is no "`writeline()`" method since it would be equivalent to calling `write()` with a single line string terminated with a NEWLINE character.

Intra-file Motion

The `seek()` method (analogous to the `fseek()` function in C) moves the file pointer to different positions within the file. The offset in bytes is given along with a *relative offset* location called *whence*. A value of 0 indicates distance from the beginning of a file (note that a position measured from the beginning of a file is also known as the *absolute offset*), a value of 1 indicates movement from the current location in the file, and a value of 2 indicates that the offset is from the end of the file. If you have used `fseek()` as a C programmer, the values 0, 1, and 2 correspond directly to the constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.

SEEK_CUR, and SEEK_END, respectively. Use of the seek() method comes to play when opening a file for read and write access. tell() is a complementary method to seek(); it tells you the current location of the file—in bytes from the beginning of the file.

Others

The close() method completes access to a file by closing it. The Python garbage collection routine will also close a file when the file object reference has decreased to zero. One way this can happen is when only one reference exists to a file, say, fp = open(), and fp is reassigned to another file object before the original file is explicitly closed. Good programming style suggests closing the file before reassignment to another file object.

The fileno() method passes back the file descriptor to the open file. This is an integer argument that can be used in lower-level operations such as those featured in the os module. The flush() method. isatty() is a Boolean built-in method that returns 1 if the file is a tty-like device and 0 otherwise. The truncate() method truncates the file to 0 or the given size bytes.

File Method Miscellany

```
filename = raw_input('Enter file name: ')
file = open(filename, 'r')
allLines = file.readlines()
file.close()
```

```
for eachLine in allLines:  
    print eachline,
```

We originally described how this program differs from most standard file access in that all the lines are read ahead of time before any display to the screen occurs. Obviously, this is not advantageous if the file is large. In those cases, it may be a good idea to go back to the tried-and-true way of reading and displaying one line at a time:

```
filename = raw_input('Enter file name: ')  
file = open(filename, 'r')  
done = 0  
while not done:  
    aLine = file.readline()  
    if aLine != "":  
        print aLine,  
    else:  
        done = 1  
file.close()
```

In this example, we do not know when we will reach the end of the file, so we create a Boolean flag `done`, which is initially set for false. When we reach the end of the file, we will reset this value to true so that the while loop will exit. We change from using `readlines()` to read all lines to `readline()`, which reads only a single line. `readline()` will return a blank line if the end of the file has been reached. Otherwise, the line is displayed to the screen.

The first highlighting output to files (rather than input)

```
filename = raw_input('Enter file name: ')
file = open(filename, 'w')

done = 0
while not done:
    aLine = raw_input("Enter a line ('.' to quit): ")
    if aLine != ".":
        file.write(aLine + '\n')
    else:
        done = 1
file.close()
```

The second performing both file input and output as well as using the seek() and tell() methods for file positioning.

```
>>> f = open('/tmp/x', 'w+')
>>> f.tell()
0
>>> f.write('test line 1\n') # add 12-char string [0–11]
>>> f.tell()
12
>>> f.write('test line 2\n') # add 12-char string [12–23]
>>> f.tell() # tell us current file location (end)
24
```

>>> f.seek(-12, 1)	# move back 12 bytes
>>> f.tell()	# to beginning of line 2
12	
>>> f.readline()	
'test line 2\012'	
>>> f.seek(0, 0)	# move back to beginning
>>> f.readline()	
'test line 1\012'	
>>> f.tell()	# back to line 2 again
12	
>>> f.readline()	
'test line 2\012'	
>>> f.tell()	# at the end again
24	
>>> f.close()	# close file

All the built-in methods for file objects:

Table 9.3. Methods for File Objects	
<i>File Object Method</i>	<i>Operation</i>
<code>file.close()</code>	close <i>file</i>
<code>file.fileno()</code>	return integer file descriptor (FD) for <i>file</i>
<code>file.flush()</code>	flush internal buffer for <i>file</i>
<code>file.isatty()</code>	return 1 if <i>file</i> is a tty-like device, 0 otherwise
<code>file.read (size=-1)</code>	read all or <i>size</i> bytes of file as a string and return it
<code>file.readinto(buf, size)</code> [a]	read <i>size</i> bytes from <i>file</i> into buffer <i>buf</i>
<code>file.readline()</code>	read and return one line from <i>file</i> (includes trailing "\n")
<code>file.readlines()</code>	read and returns all lines from <i>file</i> as a list (includes all trailing "\n" characters)
<code>file.seek(off, whence)</code>	move to a location within <i>file</i> , <i>off</i> bytes offset from <i>whence</i> (0 == beginning of file, 1 == current location, or 2 == end of file)
<code>file.tell()</code>	return current location within <i>file</i>
<code>file.truncate(size=0)</code>	truncate <i>file</i> to 0 or <i>size</i> bytes
<code>file.write(str)</code>	write string <i>str</i> to <i>file</i>
<code>file.writelines(list)</code>	write <i>list</i> of strings to <i>file</i>

File Built-in Attributes

File objects also have data attributes in addition to its methods. These attributes hold auxiliary data related to the file object they belong to, such as the file name (*file.name*), the mode with which the file was opened (*file.mode*), whether the file is closed (*file.closed*), and a flag indicating whether an additional space character needs to be displayed before successive data items when using the **print** statement (*file.softspace*).

Table 9.4. Attributes for File Objects	
<i>File Object Attribute</i>	<i>Description</i>
<i>file.closed</i>	1 if <i>file</i> is closed, 0 otherwise
<i>file.mode</i>	access mode with which <i>file</i> was opened
<i>file.name</i>	name of <i>file</i>
<i>file.softspace</i>	0 if space explicitly required with print , 1 otherwise; rarely used by the programmer—generally for internal use only

Standard Files

There are generally three standard files which are made available to you when your program starts. These are standard input (usually the keyboard), standard output (buffered output to the monitor or display), and standard error (unbuffered output to the screen). (The "buffered" or "unbuffered" output refers to that third argument to `open()`). These files are named `stdin`, `stdout`, and `stderr` and take after their names from the C language. When we say these files are "available to you when your program starts," that

means that these files are pre-opened for you, and access to these files may commence once you have their file handles. Python makes these file handles available to you from the `sys` module. Once you import `sys`, you have access to these files as `sys.stdin`, `sys.stdout`, and `sys.stderr`. The **print** statement normally outputs to `sys.stdout` while the `raw_input()` built-in function receives its input from `sys.stdin`.

We will now take yet another look at the "Hello World!" program so that you can compare the similarities and differences between using **print**/`raw_input()` and directly with the file names:

print

```
print 'Hello World!'
```

sys.stdout.write()

```
import sys
```

```
sys.stdout.write('Hello World!' + '\n')
```

Notice that we have to explicitly provide the NEWLINE character to `sys.stdout`'s `write()` method. In the input examples below, we do not because `readline()` executed on `sys.stdin` preserves the readline. `raw_input()` does not, hence we will allow `print` to add its NEWLINE.

raw_input()

```
aString = raw_input('Enter a string: ')
```

```
print aString
```


sys.stdin.readline()

```
import sys
sys.stdout.write('Enter a string: ')
aString = sys.stdin.readline()
sys.stdout.write(aString)
```

Command-line Arguments

The sys module also provides access to any *command-line arguments* via the sys.argv. Command-line arguments are those arguments given to the program in addition to the script name on invocation. Historically, of course, these arguments are so named because they are given on the command-line along with the program name in a text-based environment like a Unix- or DOS-shell. However, in an IDE or GUI environment, this would not be the case. Most IDEs provide a separate window with which to enter your Those of you familiar with C programming may ask, "Where is argc?" The strings "argv" and "argc" stand for "argument count" and "argument vector," respectively. The argv variable contains an array of strings consisting of each argument from the command-line while the argc variable contains the number of arguments entered. In Python, the value for argc is simply the number of items in the sys.argv list, and the first element of the list, sys.argv[0], is always the program name. sys.argv is the list of command-line arguments

`sys.argv` is the list of command-line arguments

`len(sys.argv)` is the number of command-line arguments (a.k.a. `argc`)

Let us create a small test program called `argv.py` with the following lines:

```
import sys
print 'you entered', len(sys.argv), 'arguments...'
print 'they were:', str(sys.argv)
```

Here is an example invocation and output of this script:

```
% argv.py 76 tales 85 hawk
```








```
you entered 5 arguments...
they were: ['argv.py', '76', 'tales', '85', 'hawk']
```

File System

Access to your file system occurs mostly through the Python `os` module. This module serves as the primary interface to your operating system facilities and services from Python. The `os` module is actually a front-end to the real module that is loaded, a module that is clearly operating system-dependent. This "real" module may be one of the following: `posix` (Unix), `nt` (Windows), `mac` (Macintosh), `dos` (DOS), `os2` (OS/2), etc. You should never import those modules directly. Just import `os` and the appropriate

module will be loaded, keeping all the underlying work hidden from sight. Depending on what your system supports, you may not have access to some of the attributes which may be available in other operating system modules. In addition to managing processes and the process execution environment, the os module performs most of the major file system operations that the application developer may wish to take advantage of. These features include removing and renaming files, traversing the directory tree, and managing file accessibility. Table 9.5 lists some of the more common file or directory operations available to you from the os module. A second module that performs specific pathname operations is also available. The os.path module is accessible through the os module. Included with this module are functions to manage and manipulate file pathname components, obtain file or directory information, and make file path inquiries. Table 9.6 outlines some of the more common functions in os.path. These two modules allow for consistent access to the file system regardless of platform or operating system. The program in Example 9.1 (ospathex.py) test drives some of these functions from the os and os.path modules.

Table 9.5. OS Module File/Directory Access Functions	
os Module File/Directory Function	Operation
File Processing	
<code>remove()</code> / <code>unlink()</code>	delete file
<code>rename()</code>	rename file
<code>*stat()</code> ^{1a1}	return file statistics
<code>symlink()</code>	create symbolic link
<code>utime()</code>	update timestamp
Directories/Folders	
<code>chdir()</code>	change working directory
<code>listdir()</code>	list files in directory

<code>getcwd()</code>	return current working directory
<code>mkdir()/makedirs()</code>	create directory(ies)
<code>rmdir()/removedirs()</code>	remove directory(ies)
<i>Access/Permissions (available only on Unix  or Windows )</i>	
<code>access()</code>	verify permission modes 
<code>chmod()</code>	change permission modes  
<code>umask()</code>	set default permission modes  

^[a] includes `stat()`, `lstat()`, `xstat()`

Table 9.6. **os.path** Module Pathname Access Functions

os.path Pathname Function	Operation
Separation	
<code>basename()</code>	remove directory path and return leaf name
<code>dirname()</code>	remove leaf name and return directory path
<code>join()</code>	join separate components into single pathname
<code>split()</code>	return (<code>dirname()</code> , <code>basename()</code>) tuple
<code>splitdrive()</code>	return (<code>drivename</code> , <code>pathname</code>) tuple
<code>splittext()</code>	return (<code>filename</code> , <code>extension</code>) tuple
Information	
<code>getatime()</code>	return last file access time
<code>getmtime()</code>	return last file modification time
<code>getsize()</code>	return file size (in bytes)
Inquiry	
<code>exists()</code>	does pathname (file or directory) exist?
<code>isdir()</code>	does pathname exist and is a directory?
<code>isfile()</code>	does pathname exist and is a file?
<code>islink()</code>	does pathname exist and is a symbolic link?
<code>samefile()</code>	do both pathnames point to the same file?

```
>>> os.path.basename("z:\Python\college.txt")  
'college.txt'
```

```
>>> os.path.dirname ("z:\Python\college.txt")  
'z:\\Python'
```

```
>>> os.path.split ("z:\Python\college.txt")  
('z:\\Python', 'college.txt')
```

```
>>> os.path.splitdrive ("z:\Python\college.txt")  
('z:', '\\Python\\college.txt')
```

```
>>> os.path.splitext ("z:\Python\college.txt")  
('z:\\Python\\college', '.txt')
```

```
>>> os.path.getatime("z:\Python\college.txt")  
1299654441.0
```

```
>>> os.path.getmtime("z:\Python\college.txt")  
1299654296.0
```

```
>>> os.path.getctime("z:\Python\college.txt")
```

```
1299654296.0758307
```

```
>>> os.path.getsize("z:\Python\college.txt")
```

```
104L
```

```
>>> os.path.exists("z:\Python\college.txt")
```

```
True
```

```
>>> os.path.exists("z:\Python\sakthi.txt")
```

```
False
```

```
>>> os.path.exists("z:\Python\college.txt")
```

```
True
```

```
>>> os.path.isdir("z:\Python\college.txt")
```

```
False
```

```
>>> os.path.isdir("z:\Python")
```

```
True
```

```
>>> os.path.isfile("z:\Python\college.txt")
```

```
True
```

MODULES

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fib.py` in the current directory with the following contents:

Python – Module

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use.

A module is a Python object with arbitrarily named attributes that you can bind and reference. Simply, a module is a file consisting of Python code. A module can define functions, classes, and variables. A module can also include runnable code.

Module Types

1. Built – in Modules
2. User – defined Modules

1. Built – in Modules

1. sys

2. array

3. math

4. time

5. regex

6. marshal

7. struct

Example:

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, *hello.py*

```
def print_func( par ):  
    print "Hello : ", par  
  
    return
```

The *import* Statement:

You can use any Python source file as a module by executing an import statement in some other Python source file. *import* has the following syntax:

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.

Example:

To import the module *hello.py*, you need to put the following command at the top of the script:

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

```
# Import module hello  
import hello
```

```
# Now you can call defined function that module as follows
```

```
hello.print_func("Zara")
```

This would produce following result:

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace:

Syntax:

```
from modname import name1[, name2[, ... nameN]]
```

Example:

For example, to import the function `fibonacci` from the module `fib`, use the following statement:

```
from fib import fibonacci
```

This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib` into the global symbol table of the importing module.

The *from...import ** Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules:

When you import a module, the Python interpreter searches for the module in the following sequences:

- The current directory.

- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.

- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The *PYTHONPATH* Variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH. Here is a typical PYTHONPATH from a Windows system:

```
set PYTHONPATH=c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system:

```
set PYTHONPATH=/usr/local/lib/python
```

```

# Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```

>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

```
>>> fibo.__name__  
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module is imported somewhere.

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:


```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`).

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Note

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

Object-Oriented Programming in PYTHON

Python has been an object-oriented language from day one. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

Overview of OOP Terminology

Class: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.

Data member: A class variable or instance variable that holds data associated with a class and its objects.

Function overloading: The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects (arguments) involved.

Instance variable: A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance : The transfer of the characteristics of a class to other classes that are derived from it.

Instance: An individual object of a certain class. An object `obj` that belongs to a class `Circle`, for example, is an instance of the class `Circle`.

Instantiation : The creation of an instance of a class.

Method : A special kind of function that is defined in a class definition.

Object : A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading: The assignment of more than one function to a particular operator.

Creating Classes:

The `class` statement creates a new class definition. The name of the class immediately follows the keyword `class` followed by a colon as follows:

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

The class has a documentation string which can be access via `ClassName.__doc__`.

The `class_suite` consists of all the component statements, defining class members, data attributes, and functions.

Example:

Following is the example of a simple Python class:

```

class Employee:
    'Common base class for all employees'
    empCount = 0
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1
    def displayCount(self):
        print "Total Employee %d" % Employee.empCount
    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary

```

The variable *empCount* is a class variable whose value would be shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.

The first method `__init__()` is a special method which is called class constructor or initialization method that Python calls when you create a new instance of this class. You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you don't need to include it when you call the methods.

Creating instance objects:

To create instances of a class, you call the class using class name and pass in whatever arguments its `__init__` method accepts.

"This would create first object of Employee class"

```
emp1 = Employee("Zara", 2000)
```

"This would create second object of Employee class"

```
emp2 = Employee("Manni", 5000)
```

Accessing attributes:

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows:

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

Now putting it all together:

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

```
class Employee:
```

```
    'Common base class for all employees'
```

```
    empCount = 0
```

```

def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

def displayCount(self):
    print "Total Employee %d" % Employee.empCount

def displayEmployee(self):
    print "Name : ", self.name, ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)

"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)

emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount

```

This would produce following result:

```

Name : Zara ,Salary: 2000
Name : Manni ,Salary: 5000
Total Employee 2

```

You can add, remove, or modify attributes of classes and objects at any time:

```
emp1.age = 7      # Add an 'age' attribute.  
emp1.age = 8      # Modify 'age' attribute.  
del emp1.age      # Delete 'age' attribute.
```

Built-In Class Attributes:

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute:

<code>__dict__</code>	: Dictionary containing the class's namespace.
<code>__doc__</code>	: Class documentation string, or None if undefined.
<code>__name__</code>	: Class name.
<code>__module__</code>	: Module name in which the class is defined. This attribute is " <code>__main__</code> " in interactive mode.
<code>__bases__</code>	: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

```
print "Employee.__doc__:", Employee.__doc__  
print "Employee.__name__:", Employee.__name__  
print "Employee.__module__:", Employee.__module__  
print "Employee.__bases__:", Employee.__bases__  
print "Employee.__dict__:", Employee.__dict__
```


The self

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you do **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention, it is given the name `self`.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use `self`.

This would produce following result:

```
Employee.__doc__      : Common base class for all employees
Employee.__name__     : Employee
Employee.__module__   : __main__
Employee.__bases__    : ()
Employee.__dict__     : {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee'    : <function displayEmployee at 0xb7c8441c>,
'__doc__'            : 'Common base class for all employees',
'__init__'           : <function __init__ at 0xb7c846bc>}
```

Destroying Objects (Garbage Collection):

Python deletes unneeded objects (built-in types or class instances) automatically to free memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed garbage collection. Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes: An object's reference count increases when it's assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

You normally won't notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method `__del__()`, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any nonmemory resources used by an instance.

Example:

This `__del__()` destructor prints the class name of an instance that is about to be destroyed:

```
#!/usr/bin/python
class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y

    def __del__(self):
        class_name = self.__class__.__name__
        print class_name, "destroyed"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints the ids of the objects
                                # pt1 del pt2 del pt3
                                179
```

This would produce following result:

3083401324 3083401324 3083401324 Point destroyed

Class Inheritance:

Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name: The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax:

Derived classes are declared much like their parent class; however, a list of base classes to inherit from are given after the class name:

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    'Optional class documentation string'  
    class_suite
```

```
#!/usr/bin/python  
class Parent: # define parent class  
    parentAttr = 100  
    def __init__(self):  
        print "Calling parent constructor"
```

```

        def parentMethod(self):
            print 'Calling parent method'
        def setAttr(self, attr):
            Parent.parentAttr = attr
        def getAttr(self):
            print "Parent attribute :", Parent.parentAttr

class Child(Parent): # define child class
    def __init__(self):
        print "Calling child constructor"
    def childMethod(self):
        print 'Calling child method'

c = Child() # instance of child

c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr() # again call parent's method

```

This would produce following result:

```

Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200

```

Similar way you can drive a class from multiple parent classes as follows:

```
class A: # define your class A
    ....
class B: # define your calss B
    ....
class C(A, B): # subclass of A and B
    ....
```

You can use `issubclass()` or `isinstance()` functions to check a relationships of two classes and instances:

The **`issubclass(sub, sup)`** boolean function returns true if the given subclass **`sub`** is indeed a subclass of the superclass **`sup`**.

The **`isinstance(obj, Class)`** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of *Class*

Overriding Methods:

You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example:

```
#!/usr/bin/python
class Parent:      # define parent class
    def myMethod(self):
        print 'Calling parent method'
class Child(Parent): # define child class
    def myMethod(self):
        print 'Calling child method'

c = Child() # instance of child
c.myMethod() # child calls overridden method
```

This would produce following result:

Calling child method

Parameterless Constructor

```
class Addition:
```

```
    x = 0
```

```
    y = 0
```

```
    z = 0
```

```
    def __init__(self):
```

```
        Addition.x = int(raw_input("Enter the X Value: "))
```

```
        Addition.y = int(raw_input("Enter the Y Value: "))
```

```
    def Add(self):
```

```
        Addition.z = Addition.x + Addition.y
```

```
        print "Addition of the Two Numbers:", Addition.z
```

```
add = Addition()
```

```
add.Add();
```


Parameterized Constructor

```
class Addition:
```

```
    x = 0
```

```
    y = 0
```

```
    z = 0
```

```
    def __init__(self,a,b):
```

```
        Addition.x = a
```

```
        Addition.y = b
```

```
    def Add(self):
```

```
        Addition.z = Addition.x + Addition.y
```

```
        print "Addition of the Two Numbers:", Addition.z
```

```
x = int(raw_input("Enter the X Value: "))
```

```
y = int(raw_input("Enter the Y Value: "))
```

```
add = Addition(x,y)
```

```
add.Add();
```

How to Call Super Class Constructor using Sub Class Constructor in Python

```
class A:  
    a = 0  
    b = 0  
  
    def __init__(self,x,y):  
        A.a = x  
        A.b = y  
  
    def display_1(self):  
        print "The X Value is:",A.a  
        print "The Y Value is:",A.b
```

```
class B(A):  
    c = 0  
  
    def __init__(self,x,y,z):  
        A.__init__(self,x,y)  
        B.c = z  
  
    def display_2(self):  
        print "The Z Value is:",B.c
```

```
b = B(10,20,30)  
b.display_1()  
b.display_2()
```

EXECUTION ENVIRONMENT

There are multiple ways in Python to run a command or execute a file on disk. It all depends on what you are trying to accomplish. There are many possible scenarios during execution:

1. Remain executing within our current script
2. Create and manage a subprocess
3. Execute an external command or program
4. Execute a command which requires input
5. Invoke a command across the network
6. Execute a command creating output which requires processing
7. Execute another Python script
8. Execute a command or program in a secure environment
9. Execute a set of dynamically-generated Python statements
10. Import a Python module (and executing its top-level code)

Callable Objects

A number of Python objects are what we describe as "callable," meaning any object which can be invoked with the function operator "()". The function operator is placed immediately following the name of the callable to invoke it. For example, the function "foo" is called with "foo()". You already know this. Callables may also be invoked via functional programming interfaces such as `apply()`, `filter()`, `map()`, and `reduce()`. Python has four callable objects: functions, methods, classes, and some class instances. Keep in mind that any additional references or aliases of these objects are callable, too.

Functions

The first callable object we introduced was the function. There are three types of different function objects, the first being the Python built-in functions.

Built-in Functions (BIFs)

BIFs are generally written as extensions in C or C++, compiled into the Python interpreter, and loaded into the system as part of the first (built-in) namespace. As mentioned in previous chapters, these functions are found in the `__builtin__` module and are imported into the interpreter as the `__builtins__` module. In restricted execution modes, only a subset of these functions is available.

BIF Attribute Description

bif.__doc__ documentation string

bif.__name__ function name as a string

bif.__self__ set to None (reserved for built-in methods)

You can verify these attributes by using the `dir()` built-in function, as indicated below using the `type()` BIF as our example:

```
>>> dir(type)
['__doc__', '__name__', '__self__']
```

Internally, built-in functions are represented as the same type as built-in methods, so invoking the `type()` built-in function on a built-in function or method outputs `"builtin_function_or_method,"` as indicated in the following example:

```
>>> type(type)
<type 'builtin_function_or_method'>
```

User-defined Functions (UDFs)

The second type of function is the user-defined function. These are generally defined at the top-level part of a module and hence are loaded as part of the global namespace (once the built-in namespace has been established). Functions may also be defined in other functions;

however, the function at the innermost level does not have access to the containing function's local scope. As indicated in previous chapters, Python currently supports only two scopes: the global scope and a function's local scope. All the names defined in a function, including parameters, are part of the local namespace.

UDF Attribute Description

udf.__doc__ documentation string (also udf.func_doc)
udf.__name__ function name as a string (also udf.func_name)
udf.func_code byte-compiled code object
udf.func_defaults default argument tuple
udf.func_globals global namespace dictionary; same as calling globals(x)

from within function Internally, user-defined functions are of the type "function," as indicated in the following example by using type():

```
>>> def foo(): pass
>>> type(foo)
<type 'function'>
```

lambda Expressions (Functions named "<lambda>")

Lambda expressions are the same as user-defined functions with some minor differences. Although they yield function objects, lambda expressions are not created with the **def statement** and instead are created using the **lambda keyword**. Because lambda expressions do not provide the infrastructure for naming the code which are tied

to them, lambda expressions must be called either through functional programming interfaces or have their reference be assigned to a variable, and then they can be invoked directly or again via functional programming. This variable is merely an alias and is *not the function object's name*. Function objects created by **lambda also share all the same attributes as user-defined functions**, with the only exception resulting from the fact that they are not named; the `__name__` or `func_name` attribute is given the string "<lambda>". Using the `type()` built-in function, we show that lambda expressions yield the same function objects as user-defined functions:

```
>>> lambdaFunc = lambda x: x * 2
```

```
>>> lambdaFunc(100)
```

```
200
```

```
>>> type(lambdaFunc)
```

```
<type 'function'>
```

In the example above, we assign the expression to an alias. We can also invoke `type()` directly on a lambda expression:

```
>>> type(lambda: 1)
```

```
<type 'function'>
```

Let's take a quick look at UDF names, using `lambdaFunc` above and `foo` from the preceding subsection:

```
>>> foo.__name__  
'foo'  
>>> lambdaFunc.__name__  
'<lambda>'
```

Methods

In the previous chapter, we discovered methods, functions which are defined as part of a class— these are user-defined methods. Many Python data types such as lists and dictionaries also have methods, known as built-in methods. To further show this type of "ownership," methods are named with or represented alongside the object's name via the dotted-attribute notation.

Built-in Methods (BIMs)

We discussed in the previous section how built-in methods are similar to built-in functions. Only built-in types (BITs) have BIMs. As you can see below, the `type()` built-in function gives the same output for built-in methods as it does for built-in functions—note how we have to provide a built-in type (object or reference) in order to access a BIM:

```
>>> type([].append)  
<type 'builtin_function_or_method'>
```

Furthermore, both BIMs and BIFs share the same attributes, too. The only exception is that now the `__self__` attribute points to a Python object (for BIMs) as opposed to `None` (for BIFs):

BIM Attribute

bim.__doc__
bim.__name__
bim.__self__

Description

documentation string
function name as a string
object the method is bound to

By convention, a BIT should have the following lists of its BIMs and (built-in) attributes.

BIT Attribute

bit.__methods__
bit.__members__

Description

list of (built-in) methods
list of (built-in) data attributes

Recall that for classes and instances, their data and method attributes can be obtained by using the `dir()` built-in function with that object as the argument to `dir()`. Apparently, BITs have two attributes that list their data and method attributes. Attributes of BITs may be accessed with either a reference or an actual object, as in these examples:

```
>>> aList = ['on', 'air']
>>> aList.append('velocity')
>>> aList
['on', 'air', 'velocity']
>>> aList.insert(2, 'speed')
>>> aList
['on', 'air', 'speed', 'velocity']
>>>
>>> [].__methods__
```

```
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']193
```

```
>>> [3, 'headed', 'knight'].pop()
'knight'
```

User-defined Methods (UDMs)

User-defined methods are contained in class definitions and are merely "wrappers" around standard functions, applicable only to the class they are defined for. They may also be called by subclass instances if not overridden in the subclass definition. As explained in the previous chapter, UDMs are associated with class objects (unbound methods), but can be invoked only with class instances (bound methods). Regardless of whether they are bound or not, all UDMs are of the same type, "instance method," as seen in the following calls to type():

```
>>> class C: # define class
... def foo(self): pass # define UDM
...
>>> c = C() # instantiation
>>> type(C.foo) # type of unbound method
<type 'instance method'>
>>> type(c.foo) # type of bound method
<type 'instance method'>
```

UDMs have the following attributes:

UDM Attribute	Description
<i>udm.__doc__</i>	<i>documentation string</i>
<i>udm.__name__</i>	<i>method name as a string</i>
<i>udm.im_class</i>	<i>class which method is associated with</i>
<i>udm.im_func f</i>	<i>unction object for method (see UDFs)</i>
<i>udm.im_self</i>	<i>associated instance if bound, None if</i>

unbound

Executable Object Statements and Built-in Functions

Python provides a number of built-in functions supporting callables and executable objects, including the **exec statement**. These functions let the programmer execute code objects as well as generate them using the compile () built-in function

Built-in Function or Statement

Description

callable(*obj*)

determines if *obj* is callable; returns 1 if so, 0 otherwise

compile(*string*, *file*, *type*)

creates a code object from *string* of type *type*; *file* is where the code originates from (usually set to ?)

eval(*obj*, *globals*=globals(),
 locals=locals())

evaluates *obj*, which is either a expression compiled into a code object or a string expression; global and/or local namespace dictionaries may also be provided, otherwise, the defaults for the current environment will be used

exec *obj*

execute *obj*, a single Python statement or set of statements, either in code object or string format; *obj* may also be a file object (opened to a valid Python script)

input(*prompt*=")

equivalent to eval(raw_input(*prompt*="))

intern(*string*)

request intern of *string*

callable()

`callable()` is a Boolean function which determines if an object type can be invoked via the function operator (`()`). It returns 1 if the object is callable and 0 otherwise. Here are some sample objects and what `callable` returns for each type:

```
>>> callable(dir) # built-in function
1
>>> callable(1) # integer
0
```

compile ()

`compile ()` is a function which allows the programmer to generate a code object on the fly, that is, during run-time. These objects can then be executed or evaluated using the **exec** statement or `eval()` built-in function. It is important to bring up the point that both **exec** and **eval()** can take string representations of Python code to execute. When executing code given as strings, the process of byte-compiling such code must occur every time.

The `compile()` function provides a one-time byte-code compilation of code so that the precompile does not have to take place with each invocation. Naturally, this is an advantage only if the same pieces of code are executed more than once. In these cases, it is definitely better to precompile the code.

All three arguments to `compile()` are required, with the first being a string representing the Python code to compile. The second string, although required, is usually set to the empty string. This parameter represents the file name (as a string) where this code object is located or can be found. Normal usage is for `compile()` to generate a code object from a dynamically generated string of Python code—code which obviously does not read from an existing file. The last argument is a string indicating the code object type.

There are three possible values:

'eval' evaluable expression [to be used with `eval()`]

'single' single executable statement [to be used with **`exec`**]

'exec' group of executable statements [to be used with **`exec`**]

Evaluatable Expression

```
>>> eval_code = compile('100 + 200', '', 'eval')
```

```
>>> eval(eval_code)
```

```
300
```

Single Executable Statement

```
>>> single_code = compile('print "hello world!"', '', 'single')
```

```
>>> single_code
```

```
<code object ? at 120998, file "", line 0>
```

```
>>> exec single_code
```

```
hello world!
```

sys.exit() and SystemExit

The primary way to exit a program immediately and return to the calling program is the `exit()` function found in the `sys` module. The syntax for `sys.exit()` is:

```
sys.exit(status=0)
```

When `sys.exit()` is called, a `SystemExit` exception is raised. Unless monitored (in a **try statement** with an appropriate **except clause**), **this exception is generally not caught nor handled, and the** interpreter exits with the given status argument, which defaults to zero if not provided. `SystemExit` is the only exception which is not viewed as an error. It simply indicates the desire to exit Python. One popular place to use `sys.exit()` is after an error is discovered in the way a command was invoked. In particular, if the arguments are incorrect, invalid, or if there are an incorrect number of them.

sys.exitfunc()

`sys.exitfunc()` is disabled by default, but can be overridden to provide additional functionality which takes place when `sys.exit()` is called and before the interpreter exits. His function will not be passed any arguments, so you should create your function to take no arguments. As described in Beazley, if `sys.exitfunc` has already been overridden by a previously defined exit function, it is good practice to also execute *that code as part of your exit function*. Generally, exit functions are used to perform some type of shutdown activity, such as closing a file or network connection, and it is always a good idea to complete these maintenance tasks, such as releasing previously held system resources.

THE END