Chapter 9. Files and Input/Output

Chapter Topics

This chapter is intended to give you an in-depth introduction to the use of files and related input/output capabilities of Python. We introduce the file object (its built-in function, and built-in methods and attributes), review the standard files, discuss accessing the file system, hint at file execution, and briefly mention persistent storage and modules in the standard library related to "file-mania."

9.1. File Objects

File objects can be used to access not only 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-style interfaces in the same manner you would access normal files.

You will find many cases where you are dealing with "file-like" objects as you continue to develop your Python experience. Some examples include "opening a URL" for reading a Web page in real-time and launching a command in a separate process and communicating to and from it like a pair of simultaneously open files, one for write and the other for read.

The `open()` built-in function (see below) returns a file object that is then used for all succeeding operations on the file in question. There are a large number of other functions that 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 need to be sent usually involves a byte stream of some sort, whether the stream occurs as individual bytes or blocks of data.

9.2. File Built-in Functions [`open()` and `file()`]

As the key to opening file doors, the `open()` [and `file()`] built-in function provides a general interface to initiate the file input/output (I/O) process. The `open()` BIF returns a file object on a successful opening of the file or else results in an error situation. When a failure occurs, Python generates or *raises* an `IOError` exceptionwe will cover errors and exceptions in the next chapter. The basic syntax of the `open()` built-in function is:

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

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. A `'U'` mode also exists for universal NEWLINE support (see below).

Any file opened with mode `'r'` or `'U'` 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 append. All writes to files opened with `'a'` will be from end-of-file, even if you seek elsewhere during access. 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. One note regarding the binary flag: `'b'` is antiquated on all Unix systems that are POSIX-compliant (including Linux) because they treat all files as binary files, including text files. Here is an entry from the Linux manual page for `fopen()`, from which the Python `open()` function is derived:

> The mode string can also include the letter "b" either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI C3.159-1989 ("ANSI C") and has no effect; the "b" is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the "b" may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-Unix environments.)

You will find a complete list of file access modes, including the use of `'b'` if you choose to use it, in Table 9.1. If *`access_mode`* is not given, it defaults automatically to `'r'`.

Table 9.1. Access Modes for File Objects

File Mode	*Operation*
`r`	Open for read

`rU or U`[a]	Open for read with universal NEWLINE support (PEP 278)
`w`	Open for write (truncate if necessary)
`a`	Open for append (always works from 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)

[a] New in Python 2.5.

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.

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(r'c:\io.sys', 'rb') #open binary file for read
```

9.2.1. The `file()` Factory Function

2.2

The `file()` built-in function came into being in Python 2.2, during the types and classes unification. At this time, many built-in types that did not have associated built-in functions were given factory functions to create instances of those objects, i.e., `dict()`, `bool()`, `file()`, etc., to go along with those that did, i.e., `list()`, `str()`, etc.

Both `open()` and `file()` do exactly the same thing and one can be used in place of the other. Anywhere

you see references to `open()`, you can mentally substitute `file()` without any side effects whatsoever.

For foreseeable versions of Python, both `open()` and `file()` will exist side by side, performing the exact same thing. Generally, the accepted style is that you use `open()` for reading/writing files, while `file()` is best used when you want to show that you are dealing with file objects, i.e., `if instance(f, file)`.

9.2.2. Universal NEWLINE Support (UNS)

In an upcoming Core Note sidebar, we describe how certain attributes of the `os` module can help you navigate files across different platforms, all of which terminate lines with different endings, i.e., `\n`, `\r`, or `\r\n`. Well, the Python interpreter has to do the same thing, toothe most critical place is when importing modules. Wouldn't it be nicer if you just wanted Python to treat all files the same way?

That is the whole point of the UNS, introduced in Python 2.3, spurred by PEP 278. When you use the `'U'` flag to open a file, all line separators (or terminators) will be returned by Python via any file input method, i.e., `read*()`, as a NEWLINE character (`\n`) regardless of what the line-endings are. (The `'rU'` mode is also supported to correlate with the `'rb'` option.) This feature will also support files that have multiple types of line-endings. A `file.newlines` attribute tracks the types of line separation characters "seen."

If the file has just been opened and no line-endings seen, `file.newlines` is `None`. After the first line, it is set to the terminator of the first line, and if one more type of line-ending is seen, then `file.newlines` becomes a tuple containing each type seen. Note that UNS only applies to reading text files. There is no equivalent handling of file output.

UNS is turned on by default when Python is built. If you do not wish to have this feature, you can disable it by using the `--without-universal-newlines` switch when running Python's `configure` script. If you must manage the line-endings yourself, then check out the Core Note and use those `os` module attributes!

9.3. 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: input, output, movement within a file, which we will call "intra-file motion," and miscellaneous. A summary of all file methods can be found in Table 9.3. We will now discuss each category.

9.3.1. 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 integer `-1`) or *`size`* is negative, the file will be read to the end. It will be phased out and eventually removed in a future version of Python.

The `readline()` method reads one line of the open file (reads all bytes until a line-terminating character like NEWLINE is encountered). The line, including termination character(s), is returned as a string. Like `read()`, there is also an optional *`size`* option, which, if not provided, defaults to `-1`, meaning read until the line-ending characters (or EOF) are found. If present, it is possible that an incomplete line is returned if it exceeds *`size`* bytes.

The `readlines()` method does not return a string like the other two input methods. Instead, it reads all (remaining) lines and returns them as a list of strings. Its optional argument, *`sizhint`*, is a hint on the maximum size desired in bytes. If provided and greater than zero, approximately *`sizhint`* bytes in whole lines are read (perhaps slightly more to round up to the next buffer size) and returned as a list.

In Python 2.1, a new type of object was used to efficiently iterate over a set of lines from a file: the `xreadlines` object (found in the `xreadlines` module). Calling *`file`*`.xreadlines()` was equivalent to `xreadlines.xreadlines(`*`file`*`)`. Instead of reading all the lines in at once, `xreadlines()` reads in chunks at a time, and thus were optimal for use with **`for`** loops in a memory-conscious way. However, with the introduction of iterators and the new file iteration in Python 2.3, it was no longer necessary to have an `xreadlines()` method because it is the same as using `iter(`*`file`*`)`, or in a **`for`** loop, is replaced by **`for`** `eachLine` **`in`** *`file`*. Easy come, easy go.

Another odd bird is the `readinto()` method, which reads the given number of bytes into a writable buffer object, the same type of object returned by the unsupported `buffer()` built-in function. (Since `buffer()` is not supported, neither is `readinto()`.)

9.3.2. Output

The `write()` built-in method has the opposite functionality as `read()` and `readline()`. It takes a string that can consist of one or more lines of text data or a block of bytes and writes the data to the file.

The `writelines()` method operates on a list just like `readlines()`, but takes a list of strings and writes them out to a file. Line termination characters are not inserted between each line, so if desired, they must be added to the end of each line before `writelines()` is called.

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.

Core Note: Line separators are preserved

When reading lines in from a file using file input methods like `read()` *or* `readlines()`*, Python does not remove the line termination characters. It is up to the programmer. For example, the following code is fairly common to see in Python code:*

```
f = open('myFile', 'r')
data = [line.strip() for line in f.readlines()]
f.close()
```

Similarly, output methods like `write()` *or* `writelines()` *do not add line terminators for the programmer... you have to do it yourself before writing the data to the file.*

9.3.3. 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, *`whence`*. A value of `0`, the default, 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`, respectively. Use of the `seek()` method comes into 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 filein bytes from the beginning of the file.

9.3.4. File Iteration

Going through a file line by line is simple:

```
for eachLine in f:
    :
```

Inside this loop, you are welcome to do whatever you need to with `eachLine`, representing a single line of the text file (which includes the trailing line separators).

Before Python 2.2, the best way to read in lines from a file was using *`file`*`.readlines()` to read in all the data, giving the programmer the ability to free up the file resource as quickly as possible. If that was not a concern, then programmers could call *`file`*`.readline()` to read in one line at a time. For a brief

time, *file*.xreadlines() was the most efficient way to read in a file.

Things all changed in 2.2 when Python introduced iterators and file iteration. In file iteration, file objects became their own iterators, meaning that users could now iterate through lines of a file using a **for** loop without having to call read*() methods. Alternatively, the iterator next method, *file*.next() could be called as well to read in the next line in the file. Like all other iterators, Python will raise StopIteration when no more lines are available.

So remember, if you see this type of code, this is the "old way of doing it," and you can safely remove the call to readline().

```
for eachLine in f.readline():
    :
```

File iteration is more efficient, and the resulting Python code is easier to write (and read). Those of you new to Python now are getting all the great new features and do not have to worry about the past.

9.3.5. 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. It is possible to lose output data that is buffered if you do not explicitly close a file.

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, i.e., os.read().

Rather than waiting for the (contents of the) output buffer to be written to disk, calling the flush() method will cause the contents of the internal buffer to be written (or flushed) to the file immediately. isatty() is a Boolean built-in method that returns true if the file is a tty-like device and False otherwise. The TRuncate() method truncates the file to the size at the current file position or the given *size* in bytes.

9.3.6. File Method Miscellany

We will now reprise our first file example from Chapter 2:

```
filename = raw_input('Enter file name: ')
f = open(filename, 'r')
allLines = f.readlines()
f.close()
for eachLine in allLines:
    print eachLine,  # suppress print's NEWLINE
```

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 that case, it may be a good idea to go back to the tried-and-true way of reading and displaying one line at a time using a file iterator:

```
filename = raw_input('Enter file name: ')
f = open(filename, 'r')
for eachLine in f:
    print eachLine,
f.close()
```

Core Note: Line separators and other file system inconsistencies

One of the inconsistencies of operating systems is the line separator character that their file systems support. On POSIX (Unix family or Mac OS X) systems, the line separator is the NEWLINE (`\n`) character. For old MacOS, it is the RETURN (`\r`), and DOS and Win32 systems use both (`\r\n`). Check your operating system to determine what your line separator(s) are.

Other differences include the file pathname separator (POSIX uses "/", DOS and Windows use "\", and the old MacOS uses ":"), the separator used to delimit a set of file pathnames, and the denotations for the current and parent directories.

These inconsistencies generally add an irritating level of annoyance when creating applications that run on all three platforms (and more if more architectures and operating systems are supported). Fortunately, the designers of the `os` module in Python have thought of this for us. The `os` module has five attributes that you may find useful. They are listed in Table 9.2.

Table 9.2. `os` Module Attributes to Aid in Multi-platform Development

`os` Module	
Attribute	***Description***
`linesep`	String used to separate lines in a file
`sep`	String used to separate file pathname components
`pathsep`	String used to delimit a set of file pathnames
`curdir`	String name for current working directory
`pardir`	String name for parent (of current working directory)

Regardless of your platform, these variables will be set to the correct values when you import the `os` module: One less headache to worry about.

We would also like to remind you that the comma placed at the end of the **`print`** statement is to suppress the NEWLINE character that **`print`** normally adds at the end of output. The reason for this is because every line from the text file already contains a NEWLINE. `readline()` and `readlines()` do not strip off any whitespace characters in your line (see exercises.) If we omitted the comma, then your text file display would be doublespaced one NEWLINE which is part of the input and another added by the **`print`** statement.

File objects also have a `truncate()` method, which takes one optional argument, *`size`*. If it is given, then the file will be truncated to, at most, *`size`* bytes. If you call `TRuncate()` without passing in a size, it will default to the current location in the file. For example, if you just opened the file and call `TRuncate()`, your file will be effectively deleted, truncated to zero bytes because upon opening a file, the "read head" is on byte 0, which is what `tell()` returns.

Before moving on to the next section, we will show two more examples, the first highlighting output to files (rather than input), and the second performing both file input and output as well as using the `seek()` and `tell()` methods for file positioning.

```
filename = raw_input('Enter file name: ')
fobj = open(filename, 'w')
while True:
   aLine = raw_input("Enter a line ('.' to quit): ")
   if aLine != ".":
     fobj.write('%s%s' % (aLine, os.linesep)
   else:
     break
fobj.close()
```

Here we ask the user for one line at a time, and send them out to the file. Our call to the `write()` method must contain a NEWLINE because `raw_input()` does not preserve it from the user input. Because it may not be easy to generate an end-of-file character from the keyboard, the program uses the period (.) as its end-of-file character, which, when entered by the user, will terminate input and close the file.

The second example opens a file for read and write, creating the file from scratch (after perhaps truncating an already existing file). After writing data to the file, we move around within the file using `seek()`. We also use the `tell()` method to show our movement.

```
>>> 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
```

Table 9.3 lists all the built-in methods for file objects.

Table 9.3. Methods for File Objects

File Object Method	*Operation*
file.close()	Closes *file*
file.fileno()	Returns integer file descriptor (FD) for *file*
file.flush()	Flushes internal buffer for *file*
file.isatty()	Returns `true` if *file* is a tty-like device and `False` otherwise
file.next[a]( )	Returns the next line in the file [similar to *file.readline()*] or raises `StopIteration` if no more lines are available
file.read(*size*=-1)	Reads *size* bytes of file, or all remaining bytes if *size* not given or is negative, as a string and return it
file.readinto[b](*buf, size*)	Reads *size* bytes from *file* into buffer *buf* (unsupported)
file.readline(*size*=-1)	Reads and returns one line from *file*(includes line-ending characters), either one full line or a maximum of *size* characters
file.readlines(*sizhint*=0)	Reads and returns all lines from *file* as a list (includes all line termination characters); if *sizhint* given and > 0, whole lines are returned consisting of approximately *sizhint* bytes (could be rounded up to next buffer's worth)
file.xreadlines[c]( )	Meant for iteration, returns lines in *file* read as chunks in a more efficient way than `readlines()`
file.seek(*off, whence*=0)	Moves to a location within *file*, *off* bytes offset from *whence* (0 == beginning of file, 1 == current location, or 2 == end of file)
file.tell()	Returns current location within *file*

`file.truncate(size=file.tell())`	Truncates *`file`* to at most *`size`* bytes, the default being the current file location
`file.write(str)`	Writes string *`str`* to *`file`*
`file.writelines(seq)`	Writes *`seq`* of strings to *`file`*; *`seq`* should be an iterable producing strings; prior to 2.2, it was just a list of strings

[a] New in Python 2.2.

[b] New in Python 1.5.2 but unsupported.

[c] New in Python 2.1 but deprecated in Python 2.3.

PREV NEXT

9.4. File Built-in Attributes

File objects also have data attributes in addition to 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 lists these attributes along with a brief description of each.

Table 9.4. Attributes for File Objects

File Object Attribute	*Description*
file.closed	True if *file* is closed and False otherwise
file.encoding[a]	Encoding that this file useswhen Unicode strings are written to file, they will be converted to byte strings using *file*.encoding; a value of None indicates that the system default encoding for converting Unicode strings should be used
file.mode	Access mode with which *file* was opened
file.name	Name of *file*
file.newlines[a]	None if no line separators have been read, a string consisting of one type of line separator, or a tuple containing all types of line termination characters read so far
file.softspace	0 if space explicitly required with print, 1 otherwise; rarely used by the programmergenerally for internal use only

[a] New in Python 2.3.

9.5. Standard Files

There are generally three standard files that 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 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`.

Just remember that since `sys.*` are files, you have to manage the line separation characters. The **`print`** statement has the built-in feature of automatically adding one to the end of a string to output.

9.6. Command-Line Arguments

The `sys` module also provides access to any *command-line arguments* via `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 "command-line arguments." These, in turn, will be passed into the program as if you started your application from the command line.

Those of you familiar with C programming may ask, "Where is `argc`?" The names "`argc`" and "`argv`" 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. Summary:

- `sys.argv` *is the list of command-line arguments*
- `len(sys.argv)` *is the number of command-line arguments*(*aka* `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']
```

Are command-line arguments useful? Commands on Unix-based systems are typically programs that take input, perform some function, and send output as a stream of data. These data are usually sent as input directly to the next program, which does some other type of function or calculation and sends the new output to another program, and so on. Rather than saving the output of each program and potentially taking up a good amount of disk space, the output is usually "piped" into the next program as *its* input.

This is accomplished by providing data on the command line or through standard input. When a program displays or sends output to the standard output file, the result would be displayed on the screenunless that program is also "piped" to another program, in which case that standard output file is really the standard input file of the next program. I assume you get the drift by now!

Command-line arguments allow a programmer or administrator to start a program perhaps with different behavioral characteristics. Much of the time, this execution takes place in the middle of the night and runs as a batch job without human interaction. Command-line arguments and program options enable this type of functionality. As long as there are computers sitting idle at night and plenty of work to be done, there will always be a need to run programs in the background on our very

expensive "calculators."

Python has two modules to help process command-line arguments. The first (and original), `getopt` is easier but less sophisticated, while `optparse`, introduced in Python 2.3, is more powerful library and is much more object-oriented than its predecessor. If you are just getting started, we recommend `getopt`, but once you outgrow its feature set, then check out `optparse`.

PREV NEXT

9.7. 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 systemdependent. This "real" module may be one of the following: `posix` (Unix-based, i.e., Linux, MacOS X, *BSD, Solaris, etc.), `nt` (Win32), `mac` (old MacOS), `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.

Table 9.5. `os` Module File/Directory Access Functions

Function	*Description*
File Processing	
`mkfifo()/mknod()`[a]	Create named pipe/create filesystem node
`remove()/unlink()`	Delete file
`rename()/renames()`[b]	Rename file
`*stat`[c]`( )`	Return file statistics
`symlink()`	Create symbolic link
`utime()`	Update timestamp
`tmpfile()`	Create and open ('`w+b`') new temporary file
`walk()`[a]	Generate filenames in a directory tree
Directories/Folders	
`chdir()/fchdir()`[a]	Change working directory/via a file descriptor
`chroot()`[d]	Change root directory of current process
`listdir()`	List files in directory
Directories/Folders	

`getcwd()/getcwdu()`[a]	Return current working directory/same but in Unicode
`mkdir()/makedirs()`	Create directory(ies)
`rmdir()/removedirs()`	Remove directory(ies)
Access/Permissions	
`access()`	Verify permission modes
`chmod()`	Change permission modes
`chown()/lchown()`[a]	Change owner and group ID/same, but do not follow links
`umask()`	Set default permission modes
File Descriptor Operations	
`open()`	Low-level operating system open [for files, use the standard `open()` built-in functions
`read()/write()`	Read/write data to a file descriptor
`dup()/dup2()`	Duplicate file descriptor/same but to another FD
Device Numbers	
`makedev()`[a]	Generate raw device number from major and minor device numbers
`major()`[a]`/minor()`[a]	Extract major/minor device number from raw device number

[a] New in Python 2.3.

[b] New in Python 1.5.2.

[c] Includes `stat()`, `lstat()`, `xstat()`.

[d] New in Python 2.2.

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`.

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

Function	*Description*

Separation	
`basename()`	Remove directory path and return leaf name
`dirname()`	Remove leaf name and return directory path
`join()`	Join separate components into single pathname
`split()`	Return (*`dirname()`*, *`basename()`*) tuple
`splitdrive()`	Return (*`drivename`*, *`pathname`*) tuple
`splitext()`	Return (*`filename`*, *`extension`*) tuple
Information	
`getatime()`	Return last file access time
`getctime()`	Return file creation time
`getmtime()`	Return last file modification time
`getsize()`	Return file size (in bytes)
Inquiry	
`exists()`	Does pathname (file or directory) exist?
`isabs()`	Is pathname absolute?
`isdir()`	Does pathname exist and is a directory?
`isfile()`	Does pathname exist and is a file?
`islink()`	Does pathname exist and is a symbolic link?
`ismount()`	Does pathname exist and is a mount point?
`samefile()`	Do both pathnames point to the same file?

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.

Example 9.1. `os & os.path` Modules Example (`ospathex.py`)

This code exercises some of the functionality found in the `os` and `os.path` modules. It creates a test file, populates a small amount of data in it, renames the file, and dumps its contents. Other auxiliary file operations are performed as well, mostly pertaining to directory tree traversal and file pathname manipulation.

```
1 #!/usr/bin/env python
2
3 import os
4 for tmpdir in ('/tmp', r'c:\temp'):
5     if os.path.isdir(tmpdir):
6         break
7 else:
8     print 'no temp directory available'
9     tmpdir = ''
10
11 if tmpdir:
12     os.chdir(tmpdir)
13     cwd = os.getcwd()
14     print '*** current temporary directory'
15     print cwd
16
17     print '*** creating example directory...'
18     os.mkdir('example')
19     os.chdir('example')
20     cwd = os.getcwd()
21     print '*** new working directory:'
22     print cwd
23     print '*** original directory listing:'
24     print os.listdir(cwd)
25
26     print '*** creating test file...'
27     fobj = open('test', 'w')
28     fobj.write('foo\n')
29     fobj.write('bar\n')
30     fobj.close()
31     print '*** updated directory listing:'
32     print os.listdir(cwd)
33
34     print "*** renaming 'test' to 'filetest.txt'"
35     os.rename('test', 'filetest.txt')
36     print '*** updated directory listing:'
37     print os.listdir(cwd)
38
39     path = os.path.join(cwd, os.listdir (cwd)[0])
40     print '*** full file pathname'
41     print path

42     print '*** (pathname, basename) =='
43     print os.path.split(path)
44     print '*** (filename, extension) =='
45     print os.path.splitext(os.path.basename(path))
46
47     print '*** displaying file contents:'
48     fobj = open(path)
49     for eachLine in fobj:
50         print eachLine,
51     fobj.close()
52
```

```
53     print '*** deleting test file'
54     os.remove(path)
55     print '*** updated directory listing:'
56     print os.listdir(cwd)
57     os.chdir(os.pardir)
58     print '*** deleting test directory'
59     os.rmdir('example')
60     print '*** DONE'
```

The `os.path` submodule to `os` focuses more on file pathnames. Some of the more commonly used attributes are found in Table 9.6.

Running this program on a Unix platform, we get the following output:

```
$ ospathex.py
*** current temporary directory
/tmp
*** creating example directory...
*** new working directory:
/tmp/example
*** original directory listing:
[]
*** creating test file...
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing:
['filetest.txt']
*** full file pathname:
/tmp/example/filetest.txt
*** (pathname, basename) ==
('/tmp/example', 'filetest.txt')

*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE
```

Running this example from a DOS window results in very similar execution:

```
C:\>python ospathex.py
*** current temporary directory
c:\windows\temp
*** creating example directory...
*** new working directory:
c:\windows\temp\example
*** original directory listing:
```

```
[]
*** creating test file...
*** updated directory listing:
['test']
*** renaming 'test' to 'filetest.txt'
*** updated directory listing:
['filetest.txt']
*** full file pathname:
c:\windows\temp\example\filetest.txt
*** (pathname, basename) ==
('c:\\windows\\temp\\example', 'filetest.txt')
*** (filename, extension) ==
('filetest', '.txt')
*** displaying file contents:
foo
bar
*** deleting test file
*** updated directory listing:
[]
*** deleting test directory
*** DONE
```

Rather than providing a line-by-line explanation here, we will leave it to the reader as an exercise. However, we will walk through a similar interactive example (including errors) to give you a feel for what it is like to execute this script one step at a time. We will break into the code every now and then to describe the code we just encountered.

```
>>> import os
>>> os.path.isdir('/tmp')
True
>>> os.chdir('/tmp')
>>> cwd = os.getcwd()
>>> cwd
'/tmp'
```

This first block of code consists of importing the `os` module (which also grabs the `os.path` module). We verify that `'/tmp'` is a valid directory and change to that temporary directory to do our work. When we arrive, we call the `getcwd()` method to tell us where we are.

```
>>> os.mkdir('example')
>>> os.chdir('example')
>>> cwd = os.getcwd()
>>> cwd
'/tmp/example'
>>>
>>> os.listdir() # oops, forgot name
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: function requires at least one argument
>>>
>>> os.listdir(cwd) # that's better :)
[]
```

Next, we create a subdirectory in our temporary directory, after which we will use the `listdir()` method

to confirm that the directory is indeed empty (since we just created it). The problem with our first call to `listdir()` was that we forgot to give the name of the directory we want to list. That problem is quickly remedied on the next line of input.

```
>>> fobj = open('test', 'w')
>>> fobj.write('foo\n')
>>> fobj.write('bar\n')
>>> fobj.close()
>>> os.listdir(cwd)
['test']
```

We then create a test file with two lines and verify that the file has been created by listing the directory again afterward.

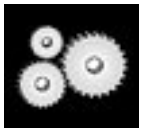
```
>>> os.rename('test', 'filetest.txt')
>>> os.listdir(cwd)
['filetest.txt']

>>>
>>> path = os.path.join(cwd, os.listdir(cwd)[0])
>>> path
'/tmp/example/filetest.txt'
>>>
>>> os.path.isfile(path)
True
>>> os.path.isdir(path)
False
>>>
>>> os.path.split(path)
('/tmp/example', 'filetest.txt')
>>>
>>> os.path.splitext(os.path.basename(path))
('filetest', '.ext')
```

This section is no doubt an exercise of `os.path` functionality, testing `join()`, `isfile()`, `isdir()` which we have seen earlier, `split()`, `basename()`, and `splitext()`. We also call the `rename()` function from `os`. Next, we display the file, and finally, we delete the temporary files and directories:

```
>>> fobj = open(path)
>>> for eachLine in fobj:
...     print eachLine,
...
foo
bar
>>> fobj.close()
>>> os.remove(path)
>>> os.listdir(cwd)
[]
>>> os.chdir(os.pardir)
>>> os.rmdir('example')
```

Core Module(S): `os` (and `os.path`)



As you can tell from our lengthy discussion above, the `os` *and* `os.path` *modules provide different ways to access the file system on your computer. Although our study in this chapter is restricted to file access only, the* `os` *module can do much more. It lets you manage your process environment, contains provisions for low-level file access, allows you to create and manage new processes, and even enables your running Python program to "talk" directly to another running program. You may find yourself a common user of this module in no time. Read more about the* `os` *module in Chapter 14.*

PREV NEXT

9.8. File Execution

Whether we want to simply run an operating system command, invoke a binary executable, or another type of script (perhaps a shell script, Perl, or Tcl/Tk), this involves executing another file somewhere else on the system. Even running other Python code may call for starting up another Python interpreter, although that may not always be the case. In any regard, we will defer this subject to Chapter 14, "Execution Environment." Please proceed there if you are interested in how to start other programs, perhaps even communicating with them, and for general information regarding Python's execution environment.

9.9. Persistent Storage Modules

In many of the exercises in this text, user input is required. After many iterations, it may be somewhat frustrating being required to enter the same data repeatedly. The same may occur if you are entering a significant amount of data for use in the future. This is where it becomes useful to have persistent storage, or a way to archive your data so that you may access them at a later time instead of having to re-enter all of that information. When simple disk files are no longer acceptable and full relational database management systems (RDBMSs) are overkill, simple persistent storage fills the gap. The majority of the persistent storage modules deals with storing strings of data, but there are ways to archive Python objects as well.

9.9.1. `pickle` *and* `marshal` *Modules*

Python provides a variety of modules that implement minimal persistent storage. One set of modules (`marshal` and `pickle`) allows for pickling of Python objects. Pickling is the process whereby objects more complex than primitive types can be converted to a binary set of bytes that can be stored or transmitted across the network, then be converted back to their original object forms. Pickling is also known as flattening, serializing, or marshalling. Another set of modules (`dbhash/bsddb`, `dbm`, `gdbm`, `dumbdbm`) and their "manager" (`anydbm`) can provide persistent storage of Python strings only. The last module (`shelve`) can do both.

As we mentioned before, both `marshal` and `pickle` can flatten Python objects. These modules do not provide "persistent storage" per se, since they do not provide a namespace for the objects, nor can they provide concurrent write access to persistent objects. What they can do, however, is to pickle Python objects to allow them to be stored or transmitted. Storage, of course, is sequential in nature (you store or transmit objects one after another). The difference between `marshal` and `pickle` is that `marshal` can handle only simple Python objects (numbers, sequences, mapping, and code) while `pickle` can transform recursive objects, objects that are multi-referenced from different places, and user-defined classes and instances. The `pickle` module is also available in a turbo version called `cPickle`, which implements all functionality in C.

9.9.2. DBM-style Modules

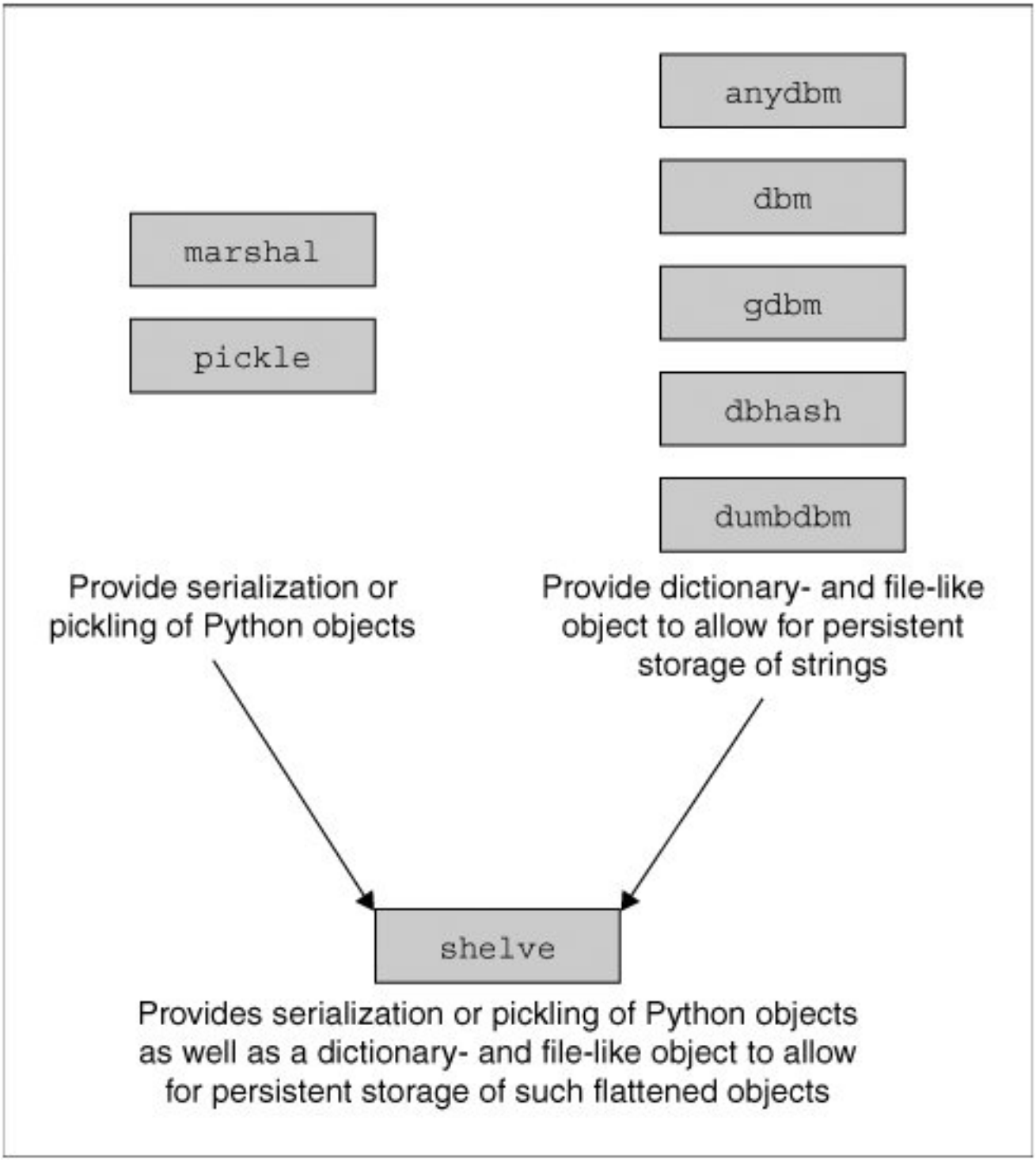
The `*db*` series of modules writes data in the traditional DBM format. There are a large number of different implementations: `dbhash/bsddb`, `dbm`, `gdbm`, and `dumbdbm`. If you are particular about any specific DBM module, feel free to use your favorite, but if you are not sure or do not care, the generic `anydbm` module detects which DBM-compatible modules are installed on your system and uses the "best" one at its disposal. The `dumbdbm` module is the most limited one, and is the default used if none of the other packages is available. These modules do provide a namespace for your objects, using objects that behave similar to a combination of a dictionary object and a file object. The one limitation of these systems is that they can store only strings. In other words, they do not serialize Python objects.

9.9.3. `shelve` *Module*

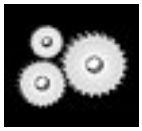
Finally, we have a somewhat more complete solution, the `shelve` module. The `shelve` module uses the `anydbm` module to find a suitable DBM module, then uses `cPickle` to perform the pickling process. The `shelve` module permits concurrent read access to the database file, but not shared read/write access. This is about as close to persistent storage as you will find in the Python standard library. There may be

other external extension modules that implement "true" persistent storage. The diagram in Figure 9-1 shows the relationship between the pickling modules and the persistent storage modules, and how the shelve object appears to be the best of both worlds.

Figure 9-1. Python modules for serialization and persistency



Core Module: `pickle` and `cPickle`



The `pickle` module allows you to store Python objects directly to a file without having to convert them to strings or to necessarily write them out as binary files using low-level file access. Instead, the `pickle` module creates a Python-only binary version that allows you to cleanly read and write objects in their entirety without having to worry about all the file details. All you need is a valid file handle, and you are ready to read or write objects from or to disk.

The two main functions in the `pickle` module are `dump()` and `load()`. The `dump()` function takes a file handle and a data object and saves the object in a format it understands to the given file. When a pickled object is loaded from disk using `load()` , it knows exactly how to restore that object to its original configuration before it was saved to disk. We recommend you take a look at `pickle` and its "smarter" brother, `shelve`, which gives you dictionary-like functionality so there is even less file overhead on your part. `cPickle` is the faster C-compiled version of `pickle`.

PREV NEXT

9.10. Related Modules

There are plenty of other modules related to files and input/output, all of which work on most of the major platforms. Table 9.7 lists some of the filerelated modules.

Table 9.7. Related File Modules

Module(s)	*Contents*
base64	Encoding/decoding of binary strings to/from text strings
binascii	Encoding/decoding of binary and ASCII-encoded binary strings
bz2[a]	Allows access to BZ2 compressed files
csv[a]	Allows access to comma-separated value files
filecmp[b].	Compares directories and files
fileinput	Iterates over lines of multiple input text files
getopt/optparse[a]	Provides command-line argument parsing/manipulation
glob/fnmatch	Provides Unix-style wildcard character matching
gzip/zlib	Reads and writes GNU zip (gzip) files (needs zlib module for compression)
shutil	Offers high-level file access functionality
c/StringIO	Implements file-like interface on top of string objects
tarfile[a]	Reads and writes TAR archive files, even compressed ones
tempfile	Generates temporary file names or files
uu	uuencode and uudecode files
zipfile[c]	Tools and utilities to read and write ZIP archive files

[a] New in Python 2.3.

[b] New in Python 2.0.

[c] New in Python 1.6.

The `fileinput` module iterates over a set of input files and reads their contents one line at a time, allowing you to iterate over each line, much like the way the Perl (< >) operator works without any provided arguments. File names that are not explicitly given will be assumed to be provided from the command-line.

The `glob` and `fnmatch` modules allow for file name pattern-matching in the good old-fashioned Unix shell-style, for example, using the asterisk (`*`) wildcard character for all string matches and the (`?`) for matching single characters.

Core Tip: Tilde (~) expansion via `os.path.expanduser()`

Although the `glob` and `fnmatch` allow for Unix-style pattern-matching, they do not enable the expansion of the tilde (home directory) character, `~` . This is handled by the `os.path.expanduser()` function. You pass in a path containing a tilde, and it returns the equivalent absolute file path. Here are two examples, in a Unix-based environment and in Win32:

```
>>> os.path.expanduser('~;/py')
'/home/wesley/py'

>>> os.path.expanduser('~;/py')
'C:\\Documents and Settings\\wesley/py'
```

In addition, Unix-flavored systems also support the " `~user` " notation indicating the home directory for a specific user. Also, note that the Win32 version does not change forward slashes to the DOS backslashes in a directory path.

The `gzip` and `zlib` modules provide direct file access to the `zlib` compression library. The `gzip` module, written on top of the `zlib` module, allows for standard file access, but provides for automatic `gzip`-compatible compression and decompression. `bz2` is like `gzip` but for bzipped files.

The `zipfile` module introduced in 1.6 allows the programmer to create, modify, and read `zip` archive files. (The `tarfile` module serves as an equivalent for TAR archive files.) In 2.3, Python was given the ability to import modules archived in zip files as well. See Section 12.5.7 for more information.

The `shutil` module furnishes high-level file access, performing such functions as copying files, copying file permissions, and recursive directory tree copying, to name a few.

The `tempfile` module can be used to generate temporary filenames and files.

In our earlier chapter on strings, we described the `StringIO` module (and its C-compiled companion `cStringIO`), and how it overlays a file interface on top of string objects. This interface includes all of the standard methods available to regular file objects.

The modules we mentioned in the Persistent Storage section above (Section 9.9) include examples of a hybrid file- and dictionary-like object.

Some other Python modules that generate file-like objects include network and file socket objects (`socket` module), the `popen*()` file objects that connect your application to other running processes (`os` and `popen2` modules), the `fdopen()` file object used in low-level file access (`os` module), and opening a network connection to an Internet Web server via its Uniform Resource Locator (URL) address (`urllib` module). Please be aware that not all standard file methods may be implemented for these objects. Likewise, they may provide functionality in addition to what is available for regular files.

Refer to the documentation for more details on these file access-related modules. In addition, you can find out more about `file()/open()`, files, file objects, and UNS at:

http://docs.python.org/lib/built-in-funcs.html

http://docs.python.org/lib/bltin-file-objects.html

http://www.python.org/doc/2.3/whatsnew/node7.html

http://www.python.org/doc/peps/pep-0278/

PREV NEXT

9.11. Exercises

9-1. *File Filtering*. Display all lines of a file, except those that *start* with a pound sign (#), the comment character for Python, Perl, Tcl, and most other scripting languages.

Extra credit: Also strip out comments that begin after the first character.

9-2. *File Access*. Prompt for a number *N* and file *F*, and display the first *N* lines of *F*.

9-3. *File Information.* Prompt for a filename and display the number of lines in that text file.

9-4. *File Access.* Write a "pager" program. Your solution should prompt for a filename, and display the text file 25 lines at a time, pausing each time to ask the user to "press a key to continue."

9-5. *Test Scores*. Update your solution to the test scores problems (Exercises 5-3 and 6-4) by allowing a set of test scores be loaded from a file. We leave the file format to your discretion.

9-6. *File Comparison*. Write a program to compare two text files. If they are different, give the line and column numbers in the files where the first difference occurs.

9-7. *Parsing Files*. Win32 users: Create a program that parses a Windows `.ini` file. POSIX users: Create a program that parses the `/etc/services` file. All other platforms: Create a program that parses a system file with some kind of structure to it.

9-8. *Module Introspection*. Extract module attribute information. Prompt the user for a module name (or accept it from the command line). Then, using `dir()` and other built-in functions, extract all its attributes, and display their names, types, and values.

9-9. *"PythonDoc."* Go to the directory where your Python standard library modules are located. Examine each `.py` file and determine whether a `__doc__` string is available for that module. If so, format it properly and catalog it. When your program has completed, it should present a nice list of those modules that have documentation strings and what they are. There should be a trailing list showing which modules do not have documentation strings (the shame list). Extra credit: Extract documentation for all classes and functions within the standard library modules.

9-10. *Home Finances*. Create a home finance manager. Your solution should be able to manage savings, checking, money market, certificate of deposit (CD), and similar accounts. Provide a menu-based interface to each account as well as operations such as deposits, withdrawals, debits, and credits. An option should be given to a user to remove transactions as well. The data should be stored to file when the user quits the application (but randomly during execution for backup purposes).

9-11. *Web site Addresses.*

a.

Write a URL bookmark manager. Create a text-driven menu-based application that allows the user to add, update, or delete entries. Entries include a site name, Web site URL address, and perhaps a one-line description (optional). Allow search functionality so that a search "word" looks through both names and URLs for possible matches. Store the data to a disk file when the user quits the application, and load up the data when the user restarts.

b.

(b) Upgrade your solution to part (a) by providing output of the bookmarks to a legible and syntactically correct HTML file (`.htm` or `.html`) so that users can then point their browsers to this output file and be presented with a list of their bookmarks. Another feature to implement is allowing the creation of "folders" to allow grouping of related bookmarks. Extra credit: Read the literature on regular expressions and the Python `re` module. Add regular expression validation of URLs that users enter into their databases.

9-12. *Users and Passwords.*

Do Exercise 7-5, which keeps track of usernames and passwords. Update your code to support a "last login time" (7-5a). See the documentation for the time module to obtain timestamps for when users "log in" to the system.

Also, create the concept of an "administrative" user that can dump a list of all the users, their passwords (you can add encryption on top of the passwords if you wish [7-5c]), and their last login times (7-5b).

a.

The data should be stored to disk, one line at a time, with fields delimited by colons (:), e.g., "`joe:boohoo:953176591.145`", for each user. The number of lines in the file will be the number of users that are part of your system.

b.

Further update your example such that instead of writing out one line at a time, you pickle the entire data object and write that out instead. Read the documentation on the `pickle` module to find out how to flatten or serialize your

object, as well as how to perform I/O using picked objects. With the addition of this new code, your solution should take up fewer lines than your solution in part (a).

c.

Replace your login database and explicit use of `pickle` by converting your code to use `shelve` files. Your resulting source file should actually take up fewer lines than your solution to part (b) because some of the maintenance work is gone.

9-13. *Command-Line Arguments*.

a.

What are they, and why might they be useful?

b.

Write code to display the command-line arguments which were entered.

9-14. *Logging Results*. Convert your calculator program

(Exercise 5-6) to take input from the command line, i.e.,

```
$ calc.py 1 + 2
```

Output the result only. Also, write each expression and result to a disk file. Issuing a command of...

```
$ calc.py print
```

... will cause the entire contents of the "register tape" to be dumped to the screen and file reset/truncated. Here is an example session:

```
$ calc.py 1 + 2
3
$ calc.py 3 ^ 3
27
$ calc.py print
1 + 2
3
3 ^ 3
27
$ calc.py print
$
```

Extra credit: Also strip out comments that begin after the first character.

9-15. *Copying Files*. Prompt for two filenames (or better yet, use command-line arguments). The contents of the first file should be copied to the second file.

9-16. *Text Processing*. You are tired of seeing lines on your e-mail wrap because people type lines that are too long for your mail reader application. Create a program to scan a text file for all lines longer than 80 characters. For each of the offending lines, find the closest word before 80 characters and break the line there, inserting the remaining text to the next line (and pushing the previous next line down one). When you are done, there should be no lines longer than 80 characters.

9-17. *Text Processing*. Create a crude and elementary text file editor. Your solution is menu-driven, with the following options:

1. create file [prompt for filename and any number of lines of input],
2. display file [dump its contents to the screen],
3. edit file (prompt for line to edit and allow user to make changes),
4. save file, and
5. quit.

9-18. *Searching Files*. Obtain a byte value (0-255) and a filename. Display the number of times that byte appears in the file.

9-19. *Generating Files*. Create a sister program to the previous problem. Create a binary data file with random bytes, but one particular byte will appear in that file a set number of times. Obtain the following three values:

1. a byte value (0-255),
2. the number of times that byte should appear in the data file, and
3. the total number of bytes that make up the data file.

Your job is to create that file, randomly scatter the requested byte across the file, ensure that there are no duplicates, the file contains exactly the number of occurrences that byte was requested for, and that the resulting data file is exactly the size requested.

9-20. *Compressed Files*. Write a short piece of code that will compress and decompress gzipped or bzipped files. Confirm your solution works by using the command-line `gzip` or `bzip2` programs or a GUI program like PowerArchiver, StuffIt, and/or WinZip.

9-21. *ZIP Archive Files*. Create a program that can extract files from or add files to, and perhaps creating, a ZIP archive file.

9-22. *ZIP Archive Files*. The `unzip -l` command to dump the contents of ZIP archive is boring. Create a Python script called `lszip.py` that gives additional information such as: the compressed file size, the compressed percentage of each file (by comparing the original and compressed file sizes), and a full `time.ctime()` timestamp instead of the unzip output (of just the date and HH:MM). Hint: The `date_time` attribute of an archived file does not contain enough information to feed to `time.mktime()`... it is up to you!

9-23. *TAR Archive Files*. Repeat the previous problem for TAR archive files. One difference between these two types of files is that ZIP files are generally compressed, but TAR files are not and usually require the support of `gzip` or `bzip2`. Add either type of compression support. Extra credit: Support both `gzip` and `bzip2`.

9-24. *File Transfer Between Archive Files*. Take your solutions from the previous two problems and write a program that moves files between ZIP (`.zip`) and TAR/gzip (`.tgz/.tar.gz`) or TAR/bzip2 (`.tbz/.tar.bz2`) archive files. The files may preexist; create them if necessary.

9-25. *Universal Extractor*. Create an application that will take any number of files in an archived and/or compression format, i.e., `.zip`, `.tgz`, `.tar.gz`, `.gz`, `.bz2`, `.tar.bz2`, `.tbz`, and a target directory. The program will uncompress the standalone files to the target while all archived files will be extracted into subdirectories named the same as the archive file without the file extension. For example, if the target directory was `incoming`, and the input files were `header.txt.gz` and `data.tgz`, `header.txt` will be extracted to `incoming` while the files in `data.tgz` will be pulled out into `incoming/data`.

PREV NEXT

Chapter 10. Errors and Exceptions

Chapter Topics

Errors are an everyday occurrence in the life of a programmer. In days hopefully long since past, errors were either fatal to the program (or perhaps the machine) or produced garbage output that was not recognized as valid input by other computers or programs or by the humans who submitted the job to be run. Any time an error occurred, execution was halted until the error was corrected and code was re-executed. Over time, demand surged for a "softer" way of dealing with errors other than termination. Programs evolved such that not every error was malignant, and when they did happen, more diagnostic information was provided by either the compiler or the program during runtime to aid the programmer in solving the problem as quickly as possible. However, errors are errors, and any resolution usually took place after the program or compilation process was halted. There was never really anything a piece of code could do but exit and perhaps leave some crumbs hinting at a possible causeuntil *exceptions* and *exception handling* came along.

Although we have yet to cover classes and object-oriented programming in Python, many of the concepts presented here involve classes and class instances.[1] We conclude the chapter with an optional section on how to create your own exception classes.

[1] As of Python 1.5, all standard exceptions are implemented as classes. If new to classes, instances, and other object-oriented terminology, the reader should see Chapter 13 for clarification.

This chapter begins by exposing the reader to exceptions, exception handling, and how they are supported in Python. We also describe how programmers can generate exceptions within their code. Finally, we reveal how programmers can create their own exception classes.

10.1. What Are Exceptions?

10.1.1. Errors

Before we get into detail about what exceptions are, let us review what errors are. In the context of software, errors are either syntactical or logical in nature. Syntax errors indicate errors with the construct of the software and cannot be executed by the interpreter or compiled correctly. These errors must be repaired before execution can occur.

Once programs are semantically correct, the only errors that remain are logical. Logical errors can either be caused by lack of or invalid input, or, in other cases, by the inability of the logic to generate, calculate, or otherwise produce the desired results based on the input. These errors are sometimes known as domain and range failures, respectively.

When errors are detected by Python, the interpreter indicates that it has reached a point where continuing to execute in the current flow is no longer possible. This is where exceptions come into the picture.

10.1.2. Exceptions

Exceptions can best be described as action that is taken outside of the normal flow of control because of errors. This action comes in two distinct phases: The first is the error that causes an exception to occur, and the second is the detection (and possible resolution) phase.

The first phase takes place when an *exception condition* (sometimes referred to as *exceptional condition*) occurs. Upon detection of an error and recognition of the exception condition, the interpreter performs an operation called *raising* an exception. Raising is also known as triggering, throwing, or generating, and is the process whereby the interpreter makes it known to the current control flow that something is wrong. Python also supports the ability of the programmer to raise exceptions. Whether triggered by the Python interpreter or the programmer, exceptions signal that an error has occurred. The current flow of execution is interrupted to process this error and take appropriate action, which happens to be the second phase.

The second phase is where exception handling takes place. Once an exception is raised, a variety of actions can be invoked in response to that exception. These can range anywhere from ignoring the error, to logging the error but otherwise taking no action, performing some corrective measures and aborting the program, or alleviating the problem to allow for resumption of execution. Any of these actions represents a *continuation*, or an alternative branch of control. The key is that the programmer can dictate how the program operates when an error occurs.

As you may have already concluded, errors during runtime are primarily caused by external reasons, such as poor input, a failure of some sort, etc. These causes are not under the direct control of the programmer, who can anticipate only a few of the errors and code the most general remedies.

Languages like Python, which support the raising andmore importantlythe handling of exceptions, empower the developer by placing them in a more direct line of control when errors occur. The programmer not only has the ability to detect errors, but also to take more concrete and remedial actions when they occur. Due to the ability to manage errors during runtime, application robustness is increased.

Exceptions and exception handling are not new concepts. They are also present in Ada, Modula-3, C++,

Eiffel, and Java. The origins of exceptions probably come from operating systems code that handles exceptions such as system errors and hardware interruptions. Exception handling as a software tool made its debut in the mid-1960s with PL/1 being the first major programming language that featured exceptions. Like some of the other languages supporting exception handling, Python is endowed with the concepts of a "try" block and "catching" exceptions and, in addition, provides for more "disciplined" handling of exceptions. By this we mean that you can create different handlers for different exceptions, as opposed to a general "catch-all" code where you may be able to detect the exception that occurred in a post-mortem fashion.

PREV NEXT

10.2. Exceptions 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 containing 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.

Let us take a look at some exceptions now.

`NameError:` ***attempt to access an undeclared variable***

```
>>> foo
Traceback (innermost last):
 File "<stdin>", line 1, in ?
NameError: name 'foo' is not defined
```

`NameError` indicates access to an uninitialized variable. The offending identifier was not found in the Python interpreter's symbol table. We will be discussing *namespaces* in the next two chapters, but as an introduction, regard them as "address books" linking names to objects. Any object that is accessible should be listed in a namespace. Accessing a variable entails a search by the interpreter, and if the name requested is not found in any of the namespaces, a *`NameError`* exception will be generated.

`ZeroDivisionError:` ***division by any numeric zero***

```
>>> 1/0
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Our example above used floats, but in general, any numeric division-by-zero will result in a *`ZeroDivisionError`* exception.

`SyntaxError:` ***Python interpreter syntax error***

```
>>> for
  File "<string>", line 1
    for
       ^
SyntaxError: invalid syntax
```

`SyntaxError` exceptions are the only ones that do not occur at run-time. They indicate an improperly constructed piece of Python code which cannot execute until corrected. These errors are generated at compile-time, when the interpreter loads and attempts to convert your script to Python bytecode. These may also occur as a result of importing a faulty module.

`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
```

`IndexError` is raised when attempting to access an index that is outside the valid range of a sequence.

`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
```

Mapping types such as dictionaries depend on keys to access data values. Such values are not retrieved if an incorrect/nonexistent key is requested. In this case, a `KeyError`is raised to indicate such an incident has occurred.

`IOError:` ***input/output error***

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

Attempting to open a nonexistent disk file is one example of an operating system input/output (I/O) error. Any type of I/O error raises an `IOError` exception.

`AttributeError:` ***attempt to access an unknown object attribute***

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

In our example, we stored a value in `myInst.bar`, the `bar` attribute of instance `myInst`. Once an attribute has been defined, we can access it using the familiar dotted-attribute notation, but if it has not, as in

our case with the `foo` (non-)attribute, an `AttributeError` occurs.

PREV NEXT

10.3. Detecting and Handling Exceptions

Exceptions can be detected by incorporating them as part of a **try** statement. Any code suite of a **TRy** statement will be monitored for exceptions.

There are two main forms of the **TRy** statement: **TRy-except** and **try-finally**. These statements are mutually exclusive, meaning that you pick only one of them. A **try** statement can be accompanied by one or more **except** clauses, exactly one **finally** clause, or a hybrid **try-except-finally** combination.

try-except statements allow one to detect and handle exceptions. There is even an optional **else** clause for situations where code needs to run only when no exceptions are detected. Meanwhile, **TRy-finally** statements allow only for detection and processing of any obligatory cleanup (whether or not exceptions occur), but otherwise have no facility in dealing with exceptions. The combination, as you might imagine, does both.

10.3.1. `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 is given below. It consists of the keywords along with the **try** and **except** blocks (`try_suite` and `except_suite`) as well as optionally saving the **reason** of failure:

```
try:
    try_suite       # watch for exceptions here
except Exception[, reason]:
    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', 'r')
... except IOError, e:
...     print 'could not open file:', e
...
could not open file: [Errno 2] No such file or directory
```

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 examplea minor illustration of the power of exception handling. So what is really happening codewise?

During runtime, the interpreter attempts to execute all the code within the **`try`** statement. If an exception does not occur when the code block has completed, execution resumes past the **`except`** statement. When the specified exception named on the **`except`** statement does occur, we save the reason, and control flow immediately continues in the handler (all remaining code in the **`TRy`** clause is skipped) where we display our error message along with the cause of the error.

In our example above, we are catching only `IOError` exceptions. Any other exception will not be caught with the handler we specified. If, for example, you want to catch an `OSError`, you have to add a handler for that particular exception. We will elaborate on the **`try-except`** syntax more as we progress further in this chapter.

Core Note: Skipping code, continuation, and upward propagation

*The remaining code in the **`try`** suite from the point of the exception is never reached (hence never executed). Once an exception is raised, the race is on to decide on the continuing flow of control. The remaining code is skipped, and the search for a handler begins. If one is found, the program continues in the handler*.

If the search is exhausted without finding an appropriate handler, the exception is then propagated to the caller's level for handling, meaning the stack frame immediately preceding the current one. If there is no handler at the next higher level, the exception is yet again propagated to its caller. If the top level is reached without an appropriate handler, the exception is considered `unhandled`, and the Python interpreter will display the traceback and exit.

10.3.2. Wrapping a Built-in Function

We will now present an interactive examplestarting with the bare necessity of detecting an error, then building continuously on what we have to further improve the robustness of our code. The premise is in detecting errors while trying to convert a numeric string to a proper (numeric object) representation of its value.

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('foo')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    float('foo')
ValueError: invalid literal for float(): foo
>>>
>>> float(['this is', 1, 'list'])
Traceback (innermost last):
  File "<stdin>", line 1, in ?
    float(['this is', 1, 'list'])
TypeError: float() argument must be a string or a number
```

Notice in the errors above that `float()` does not take too kindly to strings that 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 `ValueError`s, because they are the more likely culprit. `TypeError`s rarely happen since somehow a non-string must be given to `float()`.

```
def safe_float(obj):
   try:
       return float(obj)
   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(obj):
    try:
        retval = float(obj)
    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(obj):
     try:
         retval = float(obj)
     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 startnow 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 3, in ?
    retval = float(obj)
TypeError: float() argument must be a string or a number
```

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.

10.3.3. `try` Statement with Multiple `excepts`

Earlier in this chapter, we introduced the following general syntax for **`except`**:

```
except Exception[, reason]:
     suite_for_exception_Exception
```

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[, reason1]:
     suite_for_exception_Exception1
except Exception2[, reason2]:
     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. 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(obj):
    try:
         retval = float(obj)
    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
```

10.3.4. **except** Statement with Multiple Exceptions

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

```
except (Exception1, Exception2)[, reason]:
   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 (Exc1[, Exc2[, ... ExcN]])[, reason]:
   suite_for_exceptions_Exc1_to_ExcN
```

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(obj):
    try:
        retval = float(obj)
    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
```

10.3.5. Catching All Exceptions

Using the code we saw in the previous section, we are able to catch any number of specific exceptions and handle them. What about cases where we want to catch *all* exceptions? The short answer is yes, we can definitely do it. The code for doing it was significantly improved in 1.5 when exceptions became classes. Because of this, we now have an exception hierarchy to follow.

If we go all the way up the exception tree, we find `Exception` at the top, so our code will look like this:

```
try:
    :
except Exception, e:
     # error occurred, log 'e', etc.
```

Less preferred is the bare **except** clause:

```
try:
    :
except:
     # error occurred, etc.
```

This syntax is not as "Pythonic" as the other. 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 that may generate exceptions. Rather than investigating and discovering what types of errors may occur and how they may be prevented from happening, we have a catch-all that may not do the right thing.

We are not naming any specific exceptions to catchit does not give us any information about the possible errors that could happen in our `TRy` block. Another thing is that by catching all errors, you may be silently dropping important errors that really should be sent to the caller to properly take care of them. Finally, we do not have the opportunity to save the reason for the exception. Yes, you can get it through `sys.exc_ info()`, but then you would have to import `sys` and execute that functionboth of which can be avoided, especially if all we wanted was the instance telling us why the exception occurred. It is a distinct possibility that the bare exception clause will be deprecated in a future release of Python. (See also Core Style note).

One aspect of catching all exceptions that you need to be aware of is that there are several exceptions that are not due to an error condition. These two exceptions are `SystemExit` and `KeyboardInterrupt`. `SystemExit` is for when the current Python application wants to quit, and `KeyboardInterrupt` is when a user presses CTRL-C (^C) to terminate Python. These will be caught by both code snippets above when we really want to pass them upward. A typical workaround code pattern will look like this:

```
try:
    :
except (KeyboardInterupt, SystemExit):
    # user wants to quit
    raise                # reraise back to caller
except Exception:
    # handle real errors
```

A few things regarding exceptions did change in Python 2.5. Exceptions were moved to new-style classes, a new "mother of all exception" classes named `BaseException` was installed, and the exception hierarchy was switched around (very slightly) to get rid of that idiom of having to create two handlers. Both `KeyboardInterrupt` and `SystemExit` have been pulled out from being children of `Exception` to being its peers:

2.5

```
- BaseException
  |- KeyboardInterrupt
  |- SystemExit
  |- Exception
     |- (all other current built-in exceptions)
```

You can find the entire exception hierarchy (before and after these changes) in Table 10.2.

The end result is that now you do not have to write the extra handler for those two exceptions if you have a handler for just `Exception`. This code will suffice:

```
try:
    :
```

```
except Exception, e:
     # handle real errors
```

If you really want to catch all errors, you can still do that too, but use `BaseException` instead:

```
try:
    :
except BaseException, e:
     # handle all errors
```

And of course, there is the less preferred bare **`except`**.

Core Style: Do not handle and ignore all errors

*The **`try-except`** statement has been included in Python to provide a powerful mechanism for programmers to track down potential errors and perhaps to 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 Exception:     # same as 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.*

*Bottom line: Avoid using **`try-except`** around a large block of code with a **`pass`** just to hide errors. Instead, either handle specific exceptions and ignore them (**`pass`**), or handle all errors and take a specific action. Do not do both (handle all errors, ignore all errors).*

10.3.6. "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 an exception may have an *argument* or *reason* 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 reasons for 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. We have already seen it in the syntax above: to access any provided exception reason, 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[, reason]:
     suite_for_Exception_with_Argument

# multiple exceptions
except (Exception1, Exception2, ..., ExceptionN)[, reason]:
     suite_for_Exception1_to_ExceptionN_with_Argument
```

reason is a class instance containing diagnostic information from the code raising the exception. The exception arguments themselves go into a tuple that is stored as an attribute of the class instance, an instance of the exception class from which it was instantiated. In the first alternate syntax above, *reason* is an instance of the `Exception` class.

For most standard built-in exceptions, that is, exceptions derived from `StandardError`, the tuple consists of a single string indicating the cause of the error. The actual exception name serves as a satisfactory clue, but the error string enhances the meaning even more. Operating system or other environment type errors, i.e., `IOError`, will also include an operating system error number that precedes the error string in the tuple.

Whether a *reason* contains just a string or a combination of an error number and a string, calling `str` (*reason*) should present a human-readable cause of an error. However, do not lose sight that *reason* is really a class instanceyou are only getting the error information via that class's special method `__str__()`. We have a complete treatment of special methods as we explore object-oriented programming in Chapter 13.

The only caveat is that not all exceptions raised in third-party or otherwise external modules adhere to this standard protocol of error string or error number and error string. We recommend you follow such a standard when raising your own exceptions (see Core Style note).

Core Style: Follow exception argument protocol

When you raise built-in exceptions in your own code, try to follow the protocol established by the existing Python code as far as the error information that is part of the tuple passed as the exception argument. In other words, if you raise a `ValueError`, provide the same argument information as when the interpreter raises a `ValueError` exception, and so on. This helps keep the code consistent and will prevent other applications that use your module from breaking.

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)
<class 'exceptions.TypeError'>
>>>
>>> print diag
float() argument must be a string or a number
```

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 of the `TypeError` exception class. 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 that 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'
```

As we will discover in Chapter 13Classes and OOP the special instance attribute `__class__` exists for all class instances, and the `__doc__` class attribute is available for all classes that define their documentation strings.

We will now update our `safe_float()` one more time to include the exception argument, which is passed from the interpreter from within `float()`when exceptions are generated. In our last modification to `safe_float()`, we merged both the handlers for the `ValueError` and `TypeError` exceptions into one because we had to satisfy some requirement. The problem, if any, with this solution is that no clue is given as to which exception was raised or what caused the error. The only thing returned is an error string that indicated some form of invalid argument. Now that we have the exception argument, this no longer has to be the case.

Because each exception will generate its own exception argument, if we chose to return this string rather than a generic one we made up, it would provide a better clue as to the source of the problem. 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'
```

10.3.7. Using Our Wrapped Function in an Application

We will now feature `safe_float()` in a mini application that takes a credit card transaction data file (`carddata.txt`) and reads in all transactions, including explanatory strings. Here are the contents of our example `carddata.txt` file:

```
% cat carddata.txt
# carddata.txt
previous balance
25
debits
21.64
541.24
25
credits
-25
-541.24
finance charge/late fees
7.30
5
```

Our program, `cardrun.py`, is given in Example 10.1.

Example 10.1. Credit Card Transactions (`cardrun.py`)

We use safe_float() to process a set of credit card transactions given in a file and read in as strings. A log file tracks the processing.

```
1  #!/usr/bin/env python
2
3  def safe_float(obj):
4    'safe version of float()'
5    try:
6        retval = float(obj)
7    except (ValueError, TypeError), diag:
8        retval = str(diag)
9    return retval
10
11 def main():
12   'handles all the data processing'
```

```
13   log = open('cardlog.txt', 'w')
14   try:
15       ccfile = open('carddata.txt', 'r')
16   except IOError, e:
17       log.write('no txns this month\n')
18       log.close()
19       return
20
21   txns = ccfile.readlines()
22   ccfile.close()
23   total = 0.00
24   log.write('account log:\n')
25
26   for eachTxn in txns:
27       result = safe_float(eachTxn)
28       if isinstance(result, float):
29           total += result
30           log.write('data... processed\n')
31       else:
32           log.write('ignored: %s' % result)
33   print '$%.2f (new balance)' % (total)
34   log.close()
35
36   if __name__ == '__main__':
37    main()
```

Line-by-Line Explanation

Lines 39

This chunk of code contains the body of our `safe_float()` function.

Lines 1134

The core part of our application performs three major tasks: (1) read the credit card data file, (2) process the input, and (3) display the result. Lines 14-22 perform the extraction of data from the file. You will notice that there is a **`TRy-except`** statement surrounding the file open.

A log file of the processing is also kept. In our example, we are assuming the log file can be opened for write without any problems. You will find that our progress is kept by the log. If the credit card data file cannot be accessed, we will assume there are no transactions for the month (lines 16-19).

The data are then read into the `txns` (transactions) list where it is iterated over in lines 26-32. After every call to `safe_float()`, we check the result type using the `isinstance()` built-in function. In our example, we check to see if `safe_float()` returns a string or float. Any string indicates an error situation with a string that could not be converted to a number, while all other values are floats that can be added to the running subtotal. The final new balance is then displayed as the final line of the `main()` function.

Lines 3637

These lines represent the general "start only if not imported" functionality.Upon running our program, we get the following output:

```
$ cardrun.py
$58.94 (new balance)
```

Taking a peek at the resulting log file (`cardlog.txt`), we see that it contains the following log entries after `cardrun.py` processed the transactions found in `carddata.txt`:

```
$ cat cardl og.txt
account log:
ignored: invalid literal for float(): # carddata.txt
ignored: invalid literal for float(): previous balance
data... processed
ignored: invalid literal for float(): debits
data... processed
data... processed
data... processed
ignored: invalid literal for float(): credits
data... processed

data... processed
ignored: invalid literal for float(): finance charge/
late fees
data... processed
data... processed
```

10.3.8. `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 preceding 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.

10.3.9. `finally` Clause

A **`finally`** clause is one where its suite or block of code is executed regardless of whether an exception occurred or whether it was caught (or not). You may use a **`finally`** clause with **`TRy`** by itself or with **`try-except`** (with or without an **`else`** clause). The standalone **`try-finally`** is covered in the next section, so we will just focus on the latter here.

Starting in Python 2.5, you can use the **`finally`** clause (again) with **`TRy-except`** or **`try-except-else`**. We say "again" because believe it or not, it is not a new feature. This was a feature available in Python back in the early days but was removed in Python 0.9.6 (April 1992). At the time, it helped simplify the bytecode generation process and was easier to explain, and van Rossum believed that a unified **`try-except`** (**`-else`**)**`-finally`** would not be very popular anyway. How things change well over a decade later!

Here is what the syntax would look like with **`try-except-else-finally`**:

```
try:
    A
except MyException:
    B
else:
    C
finally:
    D
```

The equivalent in Python 0.9.6 through 2.4.x. is the longer:

```
try:
    try:
        A
    except MyException:
        B
    else:
        C
finally:
    D
```

Of course, in either case, you can have more than one **`except`** clause, however the syntax requires at least one `except` clause and both the **`else`** and **`finally`** clauses are optional. *`A,B,C`*, and *`D`* are suites (code blocks). The suites will execute in that order as necessary. (Note the only flows possible are *A-C-D* [normal] and *A-B-D* [exception].) The **`finally`** block will be executed whether exceptions occur in *A,B*, and/or *C*. Code written with the older idiom will continue to run, so there are no backward-compatibility problems.

10.3.10. `try-finally` Statement

An alternative is to use **`finally`** alone with **`try`**. 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. We know that the **`finally`** suite executes regardless of an exception being triggered within the **`try`** suite.

```
try:
    try_suite
finally:
    finally_suite  # executes regardless
```

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.

One place where we can add a **`TRy-finally`** statement is by improving our code in `cardrun.py` so that we catch any problems that may arise from reading the data from the `carddata.txt` file. In the current code in Example 10.1, we do not detect errors during the read phase (using `readlines()`):

```
try:
   ccfile = open('carddata.txt')
except IOError:
   log.write('no txns this month\n')

txns = ccfile.readlines()
ccfile.close()
```

It is possible for `readlines()` to fail for any number of reasons, one of which is if `carddata.txt` was a file on the network (or a floppy) that became inaccessible. Regardless, we should improve this piece of code so that the entire input of data is enclosed in the **`try`** clause:

```
try:
    ccfile = open('carddata.txt', 'r')
    txns = ccfile.readlines()
    ccfile.close()
except IOError:
    log.write('no txns this month\n')
```

All we did was to move the `readlines()` and `close()` method calls to the `TRy` suite. Although our code is more robust now, there is still room for improvement. Notice what happens if there was an error of some sort. If the open succeeds, but for some reason th `readlines()` call does not, the exception will continue with the **`except`** clause. No attempt is made to close the file. Wouldn't it be nice if we closed the file regardless of whether an error occurred or not? We can make it a reality using **`TRy-finally`**:

```
try:
    try:
        ccfile = open('carddata.txt', 'r')
        txns = ccfile.readlines()
    except IOError:
```

```
        log.write('no txns this month\n')
finally:
        ccfile.close()
```

This code snippet will attempt to open the file and read in the data. If an error occurs during this step, it is logged, and then the file is properly closed. If no errors occur, the file is still closed. (The same functionality can be achieved using the unified **try-except-finally** statement above.) An alternative implementation involves switching the **try-except** and **try-finally** clauses:

```
try:
    try:
        ccfile = open('carddata.txt', 'r')
        txns = ccfile.readlines()
    finally:
        ccfile.close()
except IOError:
     log.write('no txns this month\n')
```

The code works virtually the same with some differences. The most obvious one is that the closing of the file happens before the exception handler writes out the error to the log. This is because **finally** automatically reraises the exception.

One argument for doing it this way is that if an exception happens within the **finally** block, you are able to create another handler at the same outer level as the one we have, so in essence, be able to handle errors in both the original **try** block as well as the **finally** block. The only thing you lose when you do this is that if the **finally** block does raise an exception, you have lost context of the original exception unless you have saved it somewhere.

An argument against having the **finally** inside the **except** is that in many cases, the exception handler needs to perform some cleanup tasks as well, and if you release those resources with a **finally** block that comes before the exception handler, you have lost the ability to do so. In other words, the **finally** block is not as "final" as one would think.

One final note: If the code in the **finally** suite raises another exception, or is aborted due to a **return**, **break**, or **continue** statement, the original exception is lost and cannot be reraised.

10.3.11. `try-except-else-finally:` aka the Kitchen Sink

We can combine all the varying syntaxes that we have seen so far in this chapter to highlight all the different ways you can handle exceptions:

```
try:
   try_suite
except Exception1:
   suite_for_Exception1

except (Exception2, Exception3, Exception4):
   suite_for_Exceptions_2_3_and_4

except Exception5, Argument5:
   suite_for_Exception5_plus_argument
```

```
except (Exception6, Exception7), Argument67:
   suite_for_Exceptions6_and_7_plus_argument

except:
   suite_for_all_other_exceptions

else:
   no_exceptions_detected_suite
finally:
   always_execute_suite
```

Recall from above that using a **finally** clause combined with **TRy-except** or **try-except-else** is "new" as of Python 2.5. The most important thing to take away from this section regarding the syntax is that you must have at least one **except** clause; both the **else** and **finally** clauses are optional.

PREV NEXT

10.4. Context Management

10.4.1. `with` Statement

The unification of **`TRy-except`** and **`TRy-finally`** as described above makes programs more "Pythonic," meaning, among many other characteristics, simpler to write and easier to read. Python already does a great job at hiding things under the covers so all you have to do is worry about how to solve the problem you have. (Can you imagine porting a complex Python application into C++ or Java?!?)

Another example of hiding lower layers of abstraction is the **`with`** statement, made official as of Python 2.6. (It was introduced in 2.5 as a preview and to serve warnings for those applications using `with` as an identifier that it will become a keyword in 2.6. To use this feature in 2.5, you must import it with **`from`** `__future__` **`import`**`with_statement`.)

Like **`try-except-finally`**, the **`with`** statement, has a purpose of simplifying code that features the common idiom of using the **`TRy-except`** and **`try-finally`** pairs in tandem. The specific use that the **`with`** statement targets is when **`TRy-except`** and **`try-finally`** are used together in order to achieve the sole allocation of a shared resource for execution, then releasing it once the job is done. Examples include files (data, logs, database, etc.), threading resources and synchronization primitives, database connections, etc.

However, instead of just shortening the code and making it easier to use like **`try-except-finally`**, the **`with`** statement's goal is to remove the **`TRy`**, **`except`**, and **`finally`** keywords and the allocation and release code from the picture altogether. The basic syntax of the **`with`** statement looks like this:

```
with context_expr [as var]:
    with_suite
```

It looks quite simple, but making it work requires some work under the covers. The reason is it not as simple as it looks is because you cannot use the **`with`** statement merely with any expression in Python. It only works with objects that support what is called the *context management protocol*. This simply means that only objects that are built with "context management" can be used with a **`with`** statement. We will describe what that means soon.

Now, like any new video game hardware, when this feature was released, some folks out there took the time to develop new games for it so that you can play when you open the box. Similarly, there were already some Python objects that support the protocol. Here is a short list of the first set:

- `file`
- `decimal.Context`
- `tHRead.LockType`
- `threading.Lock`
- `threading.RLock`
- `threading.Condition`
- `tHReading.Semaphore`

- `tHReading.BoundedSemaphore`

Since files are first on the list and the simplest example, here is a code snippet of what it looks like to use a **`with`** statement:

```
with open('/etc/passwd', 'r') as f:
     for eachLine in f:
          # ...do stuff with eachLine or f...
```

What this code snippet will do is... well, this *is* Python, so you can probably already guess. It will do some preliminary work, such as attempt to open the file, and if all goes well, assign the file object to `f`. Then it iterates over each line in the file and does whatever processing you need to do. Once the file has been exhausted, it is closed. If an exception occurs either at the beginning, middle, or end of the block, then some cleanup code must be done, but the file will still be closed automatically.

Now, because a lot of the details have been pushed down and away from you, there are really two levels of processing that need to occur: First, the stuff at the user levelas in, the things you need to take care of as the user of the objectand second, at the object level. Since this object supports the context management protocol, it has to do some "context management."

10.4.2. *Context Management Protocol

Unless you will be designing objects for users of the **`with`** statement, i.e., programmers who will be using your objects to design their applications with, most Python programmers are going to be just users of the **`with`** statement and can skip this optional section.

We are not going into a full and deep discussion about context management here, but we will explain the types of objects and the functionality that are necessary to be protocol-compliant and thus be eligible to be used with the **`with`** statement.

Previously, we described a little of how the protocol works in our example with the file object. Let us elaborate some more here.

Context Expression (`context_expr`), Context Manager

When the **`with`** statement is executed, the context expression is evaluated to obtain what is called a *context manager*. The job of the context manager is to provide a context object. It does this by invoking its required `__context__()` special method. The return value of this method is the context object that will be used for this particular execution of the `with_suite`. One side note is that a context object itself can be its own manager, so *context_expr* can really be either a real context manager or a context object serving as its own manager. In the latter case, the context object also has a `__context__()` method, which returns `self`, as expected.

Context Object, `with_suite`

Once we have a context object, its `__enter__()` special method is invoked. This does all the preliminary stuff before the *`with_suite`* executes. You will notice in the syntax above that there is an optional **`as`** *`var`* piece following *`context_expr`* on the **`with`** statement line. If *`var`* is provided, it is assigned the return value of `__enter__()`. If not, the return value is thrown away. So for our file object example, its context object's `__enter__()` returns the file object so it can be assigned to `f`.

Now the *with_suite* executes. When execution of *with_suite* terminates, whether "naturally" or via exception, the context object's `__exit__()` special method is called.`__exit__()`takes three arguments. If *with_suite* terminates normally, all three parameters passed in are `None`. If an exception occurred, then the three arguments are the same three values returned when calling the `sys.exc_info()` function (see section 10.12): `type` (exception class), `value` (this exception's instance), and `traceback`, the corresponding traceback object.

It is up to you to decide how you want to handle the exception here in `__exit__()`. The usual thing to do after you are done is not to return anything from `__exit__()` or return `None` or some other Boolean `False` object. This will cause the exception to be reraised back to your user for handling. If you want to explicitly silence the exception, then return any object that has a Boolean `TRue` value. If an exception did not occur or you returned `true` after handling an exception, the program will continue on the next statement after the **`with`** clause.

Since context management makes the most sense for shared resources, you can imagine that the `__enter__()` and `__exit__()` methods will primarily be used for doing the lower-level work required to allocate and release resources, i.e., database connections, lock allocation, semaphore decrement, state management, opening/closing of files, exception handling, etc.

To help you with writing context managers for objects, there is the `contextlib` module, which contains useful functions/decorators with which you can apply over your functions or objects and not have to worry about implementing a class or separate `__context__()`, `__enter__()`, `__exit__()` special methods.

For more information or more examples of context management, check out the official Python documentation on the **`with`** statement and `contextlib` module, class special methods (related to **`with`** and contexts), PEP 343, and the "What's New in Python 2.5" document.

PREV NEXT

10.5. *Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. With the advent of exception classes, this is no longer the case. As of 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

For backward compatibility, it is possible to revert to string-based exceptions. Starting the Python interpreter with the command-line option `-X`will provide you with the standard exceptions as strings. This feature will be obsolete beginning with Python 1.6.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning. In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions within the scope of this book and have removed it. (You may find the original text in prior editions of this book.) The only point of relevance and the final thought is a caution: You may use an external or third-party module, which may still have string exceptions. String exceptions are a bad idea anyway. One reader vividly recalls seeing Linux RPM exceptions with spelling errors in the exception text.

10.6. Raising Exceptions

The interpreter was responsible for raising all of the exceptions 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.

10.6.1. raise Statement

Syntax and Common Usage

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

```
raise [SomeException [, args [, traceback]]]
```

The first argument, *SomeException*, is the name of the exception to raise. If present, it must either be a string, class, or instance (more below). *SomeException* must be given if any of the other arguments (*args* or *traceback*) are present. A list of all Python standard exceptions is given in Table 10.2.

The second expression contains optional args (aka 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 that 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 exceptionnormally a traceback object is newly created when an exception is raised. This third argument is useful if you want to reraise an exception (perhaps to point to the previous location from the current). Arguments that are absent are represented by the value None.

The most common syntax used is when *SomeException* is a class. No additional parameters are ever required, but in this case, if they are given, they 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.

More Exotic/Less Common Usage

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 or 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 instanceis 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 instanceinstead, it is a singleton or tuplethen 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.

If *`SomeException`* is an instance, then we do not need to instantiate anything. In this case, additional parameters must not be given or must be `None`. The exception type is the class that `instance`belongs to; in other words, this is equivalent to raising the class with this instance, i.e., **raise** `instance.__class__, instance`.

Use of string exceptions is deprecated in favor of exception classes, but if *`SomeException`* is a string, then it raises the exception identified by `string`, with any optional parameters (`args`) as arguments.

Finally, the **raise** statement by itself *without any parameters* is a new construct, introduced in Python 1.5, and causes the last exception raised in the current code block to be reraised. If no exception was previously raised, a `TypeError` exception will occur, because there was no previous exception to reraise.

Due to the many different valid syntax formats for **raise** (i.e., *`SomeException`* can be either a class, instance, or a string), we provide Table 10.1 to illuminate all the different ways which **raise** can be used.

Table 10.1. Using the `raise` Statement

***raise* syntax**	***Description***
raise *exclass*	Raise an exception, creating an instance of `exclass` (without any exception arguments)
raise *exclass()*	Same as above since classes are now exceptions; invoking the class name with the function call operator instantiates an instance of `exclass`, also with no arguments
raise *exclass*	Same as above, but also providing exception arguments *`args`*, which can be a single argument or a tuple
raise*exclass(args)*	Same as above
raise *exclass,args, tb*	Same as above, but provides traceback object `tb`to use
raise *exclass, instance*	Raise exception using `instance` (normally an instance of `exclass`); if `instance` is an instance of a *subclass* of `exclass`, then the new exception will be of the subclass type (not of `exclass`); if `instance` is *not* an instance of `exclass` *or* an instance of a *subclass* of `exclass`, then a new instance of `exclass` will be created with exception arguments copied from `instance`
raise*instance*	Raise exception using `instance`: the exception type is the class that instantiated `instance`; equivalent to **raise** `instance.__ class__, instance` (same as above)
raise *string*	(*Archaic*) Raises `string` exception
raise `string, args`	Same as above, but raises exception with `args`

raise `string, args, tb`	Same as above, but provides traceback object `tb` to use
raise	(*New in 1.5*) Reraises previously raised exception; if no exception was previously raised, a `TypeError`is raised

PREV NEXT

10.7. Assertions

Assertions are diagnostic predicates that 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 precompile 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.

Assertions are carried out by the **`assert`** statement, introduced back in version 1.5.

10.7.1. `assert` Statement

The **`assert`** statement evaluates a Python expression, taking no action if the assertion succeeds (similar to a **`pass`** statement), but otherwise raising 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
```

As with 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!
```

To give you a better idea of how **assert** works, imagine how the **assert** statement may be implemented in Python if written as a function. It would probably look something like this:

```
def assert(expr, args=None):
    if __debug__ and not expr:
        raise AssertionError, args
```

The first **if** statement confirms the appropriate syntax for the assert, meaning that `expr` should be an expression. We compare the type of `expr` to a real expression to verify. The second part of the function evaluates the expression and raises `AssertionError`, if necessary. The built-in variable `__debug__` is 1 under normal circumstances, 0 when optimization is requested (command-line option `-O`).

PREV NEXT

10.8. Standard Exceptions

Table 10.2 lists all of Python's current set of standard exceptions. All exceptions are loaded into the interpreter as built-ins so they are ready before your script starts or by the time you receive the interpreter prompt, if running interactively.

Table 10.2. Python Built-In Exceptions

Exception Name	*Description*
BaseException[a]	Root class for all exceptions
SystemExit[b]	Request termination of Python interpreter
KeyboardInterrupt[c]	User interrupted execution (usually by typing ^C)
Exception[d]	Root class for regular exceptions
StopIteration[e]	Iteration has no further values
GeneratorExit[a]	Exception sent to generator to tell it to quit
SystemExit[h]	Request termination of Python interpreter
StandardError[g]	Base class for all standard built-in exceptions
ArithmeticError[d]	Base class for all numeric calculation errors

FloatingPointError[d]	Error in floating point calculation
OverflowError	Calculation exceeded maximum limit for numerical type
ZeroDivisionError	Division (or modulus) by zero error (all numeric types)
AssertionError[d]	Failure of **assert** statement
AttributeError	No such object attribute
EOFError	End-of-file marker reached without input from built-in
EnvironmentError[d]	Base class for operating system environment errors
IOError	Failure of input/output operation
OSError[d]	Operating system error
WindowsError[h]	MS Windows system call failure
ImportError	Failure to import module or object
KeyboardInterrupt[f]	User interrupted execution (usually by typing ^C)
LookupError[d]	Base class for invalid data lookup errors

IndexError	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[h]	Access of an uninitialized local variable
ReferenceError[e]	Weak reference tried to access a garbage-collected object
RuntimeError	Generic default error during execution
NotImplementedError[d]	Unimplemented method
SyntaxError	Error in Python syntax
IndentationError[g]	Improper indentation
TabError[g]	Improper mixture of TABs and spaces
SystemError	Generic interpreter system error
TypeError	Invalid operation for type
ValueError	Invalid argument given

`UnicodeError`[h]	Unicode-related error
`UnicodeDecodeError`[i]	Unicode error during decoding
`UnicodeEncodeError`[i]	Unicode error during encoding
`UnicodeTranslateError`[f]	Unicode error during translation
`Warning`[j]	Root class for all warnings
`DeprecationWarning`[j]	Warning about deprecated features
`FutureWarning`[i]	Warning about constructs that will change semantically in the future
`OverflowWarning`[k]	Old warning for auto-long upgrade
`PendingDeprecationWarning`[i]	Warning about features that will be dep recated in the future
`RuntimeWarning`[j]	Warning about dubious runtime behavior
`SyntaxWarning`[j]	Warning about dubious syntax
`UserWarning`[j]	Warning generated by user code

[a] New in Python 2.5.

[b] Prior to Python 2.5, `SystemExit` subclassed `Exception`.

[c] Prior to Python 2.5, `KeyboardInterrupt` subclassed `StandardError`.

[d] New in Python 1.5, the release when class-based exceptions replaced strings.

[e] New in Python 2.2.

[h] New in Python 1.6.

[g] New in Python 2.0.

[f] New in Python 1.6.

[i] New in Python 2.3.

[j] New in Python 2.1.

[k] New in Python 2.2 but removed in Python 2.4.

All standard/built-in exceptions are derived from the root class `BaseException`. There are currently three immediate subclasses of `BaseException`: `SystemExit`, `KeyboardInterrupt`, and `Exception`. All other built-in exceptions are subclasses of `Exceptions`. Every level of indentation of an exception listed in Table 10.2 indicates one level of exception class derivation.

As of Python 2.5, all exceptions are new-style classes and are ultimately subclassed from `BaseException`. At this release, `SystemExit` and `KeyboardInterrupt` were taken out of the hierarchy for `Exception` and moved up to being under `BaseException`. This is to allow statements like **`except`** `Exception` to catch all errors and not program exit conditions.

From Python 1.5 through Python 2.4.x, exceptions were classic classes, and prior to that, they were strings. String-based exceptions are no longer acceptable constructs and are officially deprecated beginning with 2.5, where you will not be able to *raise* string exceptions. In 2.6, you cannot *catch* them.

There is also a requirement that all new exceptions be ultimately subclassed from `BaseException` so that all exceptions will have a common interface. This will transition will begin with Python 2.7 and continue through the remainder of the 2.x releases.

PREV NEXT

10.9. *Creating Exceptions

Although the set of standard exceptions is fairly wide-ranging, it may be advantageous to create your own exceptions. One situation is where you would like additional information from what a standard or module-specific exception provides. We will present two examples, both related to `IOError`.

`IOError` is a generic exception used for input/output problems, which may arise from invalid file access or other forms of communication. Suppose we wanted to be more specific in terms of identifying the source of the problem. For example, for file errors, we want to have a `FileError` exception that behaves like `IOError`, but with a name that has more meaning when performing file operations.

Another exception we will look at is related to network programming with sockets. The exception generated by the `socket` module is called `socket.error` and is not a built-in exception. It is subclassed from the generic `Exception` exception. However, the exception arguments from `socket.error` closely resemble those of `IOError` exceptions, so we are going to define a new exception called `NetworkError`, which subclasses from `IOError` but contains at least the information provided by `socket.error`.

Like classes and object-oriented programming, we have not formally covered network programming at this stage, but skip ahead to Chapter 16 if you need to.

We now present a module called `myexc.py` with our newly customized exceptions `FileError` and `NetworkError`. The code is in Example 10.2.

Example 10.2. Creating Exceptions (`myexc.py`)

This module defines two new exceptions,FileError and NetworkError, as well as reimplements more diagnostic versions of open() [myopen()] and socket.connect() [myconnect()]. Also included is a test function [test()] that is run if this module is executed directly.

```
1   #!/usr/bin/env python
2
3   import os, socket, errno, types, tempfile
4
5   class NetworkError(IOError):
6       pass
7
8   class FileError(IOError):
9       pass
10
11  def updArgs(args, newarg=None):
12      if isinstance(args, IOError):
13          myargs = []
14          myargs.extend([arg for arg in args])
15      else:
16          myargs = list(args)
17
18      if newarg:
19          myargs.append(newarg)
20
```

```
21      return tuple(myargs)
22
23  def fileArgs(file, mode, args):
24      if args[0] == errno.EACCES and \
25              'access' in dir(os):
26          perms = ''
27          permd = { 'r': os.R_OK, 'w': os.W_OK,
28              'x': os.X_OK}
29          pkeys = permd.keys()
30          pkeys.sort()
31          pkeys.reverse()
32
33          for eachPerm in 'rwx':
34              if os.access(file, permd[eachPerm]):
35                  perms += eachPerm
36              else:
37                  perms += '-'
38
39          if isinstance(args, IOError):
40              myargs = []
41              myargs.extend([arg for arg in args])
42         else:
43              myargs = list(args)
44
45          myargs[1] = "'%s' %s (perms: '%s')" % \
46              (mode, myargs[1], perms)
47
48          myargs.append(args.filename)
49
50       else:
51           myargs = args
52
53      return tuple(myargs)
54
55  def myconnect(sock, host, port):
56      try:
57          sock.connect((host, port))
58
59      except socket.error, args:
60          myargs = updArgs(args)    # conv inst2tuple
61          if len(myargs) == 1:       # no #s on some errs
62              myargs = (errno.ENXIO, myargs[0])
63
64          raise NetworkError, \
65              updArgs(myargs, host + ':' + str(port))
66
67  def myopen(file, mode='r'):
68      try:
69          fo = open(file, mode)
70      except IOError, args:
71          raise FileError, fileArgs(file, mode, args)
72
73      return fo
74
75  def testfile():
76
77      file = mktemp()
78      f = open(file, 'w')
```

```
79      f.close()
80
81      for eachTest in ((0, 'r'), (0100, 'r'),
82              (0400, 'w'), (0500, 'w')):
83          try:
84              os.chmod(file, eachTest[0])
85             f = myopen(file, eachTest[1])
86
87          except FileError, args:
88              print "%s: %s" % \
89                  (args.__class__.__name__, args)
90         else:
91              print file, "opened ok... perm ignored"
92              f.close()
93
94      os.chmod(file, 0777)    # enable all perms
95      os.unlink(file)
96
97  def testnet():
98      s = socket.socket(socket.AF_INET,
99          socket.SOCK_STREAM)
100
101     for eachHost in ('deli', 'www'):
102       try:
103          myconnect(s, 'deli', 8080)
104       except NetworkError, args:
105          print "%s: %s" % \
106              (args.__class__.__name__, args)
107
108 if __name__ == '__main__':
109     testfile()
110     testnet()
```

Lines 13

The Unix startup script and importation of the `socket`, `os`, `errno`, `types`, and `tempfile` modules help us start this module.

Lines 59

Believe it or not, these five lines make up our new exceptions. Not just one, but both of them. Unless new functionality is going to be introduced, creating a new exception is just a matter of subclassing from an already existing exception. In our case, that would be `IOError`. `EnvironmentError`, from which `IOError` is derived, would also work, but we wanted to convey that our exceptions were definitely I/O-related.

We chose `IOError` because it provides two arguments, an error number and an error string. File-related [uses `open()`] `IOError` exceptions even support a third argument that is not part of the main set of exception arguments, and that would be the filename. Special handling is done for this third argument, which lives outside the main tuple pair and has the name `filename`.

Lines 1121

The entire purpose of the `updArgs()` function is to "update" the exception arguments. What we mean here is that the original exception is going to provide us a set of arguments. We want to take these arguments and make them part of our new exception, perhaps embellishing or adding a third argument (which is not added if nothing is given`None` is a default argument, which we will study in the next chapter). Our goal is to provide the more informative details to the user so that if and when errors occur, the problems can be tracked down as quickly as possible.

Lines 2353

The `fileArgs()` function is used only by `myopen()` (see below). In particular, we are seeking error `EACCES`, which represents "permission denied." We pass all other `IOError` exceptions along without modification (lines 54 - 55). If you are curious about `ENXIO`, `EACCES`, and other system error numbers, you can hunt them down by starting at file `/usr/include/sys/errno.h` on a Unix system, or `C:\Msdev\include\Errno.h` if you are using Visual C++ on Windows.

In line 27, we are also checking to make sure that the machine we are using supports the `os.access()` function, which helps you check what kind of file permissions you have for any particular file. We do not proceed unless we receive both a permission error as well as the ability to check what kind of permissions we have. If all checks out, we set up a dictionary to help us build a string indicating the permissions we have on our file.

The Unix file system uses explicit file permissions for the user, group (more than one user can belong to a group), and other (any user other than the owner or someone in the same group as the owner) in read, write, and execute (`r', `w', `x') order. Windows supports some of these permissions.

Now it is time to build the permission string. If the file has a permission, its corresponding letter shows up in the string, otherwise a dash (-) appears. For example, a string of "`rw-`" means that you have read and write access to it. If the string reads "`r-x`", you have only read and execute access; "`---`" means no permission at all.

After the permission string has been constructed, we create a temporary argument list. We then alter the error string to contain the permission string, something that standard `IOError` exception does not provide. "Permission denied" sometimes seems silly if the system does not tell you what permissions you have to correct the problem. The reason, of course, is security. When intruders do not have permission to access something, the last thing you want them to see is what the file permissions are, hence the dilemma. However, our example here is merely an exercise, so we allow for the temporary "breach of security." The point is to verify whether or not the `os.chmod()`functions call affected file permissions the way they are supposed to.

The final thing we do is to add the filename to our argument list and return the set of arguments as a tuple.

Lines 5565

Our new `myconnect()` function simply wraps the standard socket method `connect()`to provide an `IOError` - type exception if the network connection fails. Unlike the general `socket.error` exception, we also provide the hostname and port number as an added value to the programmer.

For those new to network programming, a hostname and port number pair are analogous to an area code and telephone number when you are trying to contact someone. In this case, we are trying to contact a program running on the remote host, presumably a server of some sort; therefore, we require

the host's name and the port number that the server is listening on.

When a failure occurs, the error number and error string are quite helpful, but it would be even more helpful to have the exact host-port combination as well, since this pair may be dynamically generated or retrieved from some database or name service. That is the value-add we are bestowing on our version of `connect()`. Another issue arises when a host cannot be found. There is no direct error number given to us by the `socket.error` exception, so to make it conform to the `IOError` protocol of providing an error number-error string pair, we find the closest error number that matches. We choose `ENXIO.`

Lines 6773

Like its sibling `myconnect()`, `myopen()`also wraps around an existing piece of code. Here, we have the `open()` function. Our handler catches only `IOError`exceptions. All others will pass through and on up to the next level (when no handler is found for them). Once an `IOError` is caught, we raise our own error and customized arguments as returned from `fileArgs()`.

Lines 7595

We shall perform the file testing first, here using the `testfile()` function. In order to begin, we need to create a test file that we can manipulate by changing its permissions to generate permission errors. The `tempfile` module contains code to create temporary file names or temporary files themselves. We just need the name for now and use our new `myopen()` function to create an empty file. Note that if an error occurred here, there would be no handler, and our program would terminate fatallythe test program should not continue if we cannot even *create* a test file.

Our test uses four different permission configurations. A zero means no permissions at all, 0100 means execute-only, 0400 indicates read-only, and 0500 means read- and execute-only (0400 + 0100). In all cases, we will attempt to open a file with an invalid mode. The `os.chmod()`function is responsible for updating a file's permission modes. (Note: These permissions all have a leading zero in front, indicating that they are octal [base 8] numbers.)

If an error occurs, we want to display diagnostic information similar to the way the Python interpreter performs the same task when uncaught exceptions occur, and that is giving the exception name followed by its arguments. The `__class__` special variable represents the class object from which an instance was created. Rather than displaying the entire class name here (`myexc.FileError`), we use the class object's `__name__` variable to just display the class name (`FileError`), which is also what you see from the interpreter in an unhandled error situation. Then the arguments that we arduously put together in our wrapper functions follow.

If the file opened successfully, that means the permissions were ignored for some reason. We indicate this with a diagnostic message and close the file. Once all tests have been completed, we enable all permissions for the file and remove it with the `os.unlink()` function. (`os.remove()` is equivalent to `os.unlink()`.)

Lines 97106

The next section of code (`testnet()`) tests our `NetworkError` exception. A socket is a communication endpoint with which to establish contact with another host. We create such an object, then use it in an attempt to connect to a host with no server to accept our connect request and a host not on our network.

Lines 108110

We want to execute our `test*()` functions only when invoking this script directly, and that is what the code here does. Most of the scripts given in this text utilize the same format.

Running this script on a Unix-flavored box, we get the following output:

```
$myexc.py
FileError: [Errno 13] 'r' Permission denied (perms: '---'):
 '/usr/tmp/@18908.1'
FileError: [Errno 13] 'r' Permission denied (perms: '--x'):
 '/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r--'):
 '/usr/tmp/@18908.1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
 '/usr/tmp/@18908.1'
NetworkError: [Errno 146] Connection refused: 'deli:8080'
NetworkError: [Errno 6] host not found: 'www:8080'
```

The results are slightly different on a Win32 machine:

```
D:\python> python myexc.py
C:\WINDOWS\TEMP\~-195619-1 opened ok... perms ignored
C:\WINDOWS\TEMP\~-195619-1 opened ok... perms ignored
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
 'C:\\WINDOWS\\TEMP\\~-195619-1'
FileError: [Errno 13] 'w' Permission denied (perms: 'r-x'):
 'C:\\WINDOWS\\TEMP\\~-195619-1'
NetworkError: [Errno 10061] winsock error: 'deli:8080'
NetworkError: [Errno 6] host not found: 'www:8080'
```

You will notice that Windows does not support read permissions on files, which is the reason why the first two file open attempts succeeded. Your mileage may vary (YMMV) on your own machine and operating system.

10.10. Why Exceptions (Now)?

There is no doubt that errors will be around as long as software is around. The difference in today's fast-paced computing world is that our execution environments have changed, and so has our need to adapt error-handling to accurately reflect the operating context of the software that we develop. Modern-day applications generally run as self-contained graphical user interfaces (GUIs) or in a client/server architecture such as the Web.

The ability to handle errors at the application level has become even more important recently in that users are no longer the only ones directly running applications. As the Internet and online electronic commerce become more pervasive, Web servers will be the primary users of application software. This means that applications cannot just fail or crash outright anymore, because if they do, system errors translate to browser errors, and these in turn lead to frustrated users. Losing eyeballs means losing advertising revenue and potentially significant amounts of irrecoverable business.

If errors do occur, they are generally attributed to some invalid user input. The execution environment must be robust enough to handle the application-level error and be able to produce a user-level error message. This must translate to a "non-error" as far as the Web server is concerned because the application must complete successfully, even if all it does is return an error message to present to the user as a valid Hypertext Markup Language (HTML) Web page displaying the error.

If you are not familiar with what I am talking about, does a plain Web browser screen with the big black words saying, "Internal Server Error" sound familiar? How about a fatal error that brings up a pop-up that declares "Document contains no data"? As a user, do either of these phrases mean anything to you? No, of course not (unless you are an Internet software engineer), and to the average user, they are an endless source of confusion and frustration. These errors are a result of a failure in the execution of an application. The application either returns invalid Hypertext Transfer Protocol (HTTP) data or terminates fatally, resulting in the Web server throwing its hands up into the air, saying, "I give up!"

This type of faulty execution should not be allowed, if at all possible. As systems become more complex and involve more apprentice users, additional care should be taken to ensure a smooth user application experience. Even in the face of an error situation, an application should terminate successfully, as to not affect its execution environment in a catastrophic way. Python's exception handling promotes mature and correct programming.

10.11. Why Exceptions at All?

If the above section was not motivation enough, imagine what Python programming might be like without program-level exception handling. The first thing that comes to mind is the loss of control client programmers have over their code. For example, if you created an interactive application that allocates and utilizes a large number of resources, if a user hit ^C or other keyboard interrupt, the application would not have the opportunity to perform cleanup, resulting in perhaps loss of data or data corruption. There is also no mechanism to take alternative action such as prompting the users to confirm whether they really want to quit or if they hit the Control key accidentally.

Another drawback would be that functions would have to be rewritten to return a "special" value in the face of an error situation, for example, `None`. The engineer would be responsible for checking each and every return value from a function call. This may be cumbersome because you may have to check return values, which may not be of the same type as the object you are expecting if no errors occurred. And what if your function wants to return `None` as a valid data value? Then you would have to come up with another return value, perhaps a negative number. We probably do not need to remind you that negative numbers may be valid in a Python context, such as an index into a sequence. As a programmer of application programmer interfaces (APIs), you would then have to document every single return error your users may encounter based on the input received. Also, it is difficult (and tedious) to propagate errors (and reasons) of multiple layers of code.

There is no simple propagation like the way exceptions do it. Because error data needs to be transmitted upwards in the call hierarchy, it is possible to misinterpret the errors along the way. A totally unrelated error may be stated as the cause when in fact it had nothing to do with the original problem to begin with. We lose the bottling-up and safekeeping of the original error that exceptions provide as they are passed from layer to layer, not to mention completely losing track of the data we were originally concerned about! Exceptions simplify not only the code, but the entire error management scheme, which should not play such a significant role in application development. And with Python's exception handling capabilities, it does not have to.

10.12. Exceptions and the sys Module

An alternative way of obtaining exception information is by accessing the `exc_info()` function in the `sys` module. This function provides a 3-tuple of information, more than what we can achieve by simply using only the exception argument. Let us see what we get using `sys.exc_info()`:

```
>>> try:
...     float('abc123')
... except:
...     import sys
...        exc_tuple = sys.exc_info()
...
>>> print exc_tuple
(<class exceptions.ValueError at f9838>, <exceptions.
ValueError instance at 122fa8>,
<traceback object at 10de18>)
>>>

>>> for eachItem in exc_tuple:
...     print eachItem
...
exceptions.ValueError
invalid literal for float(): abc123
<traceback object at 10de18>
```

What we get from `sys.exc_info()` in a tuple are:

- `exc_type`: exception class object
- `exc_value`: (this) exception class instance object
- `exc_traceback`: traceback object

The first two items we are familiar with: the actual exception class and this particular exception's instance (which is the same as the exception argument which we discussed in the previous section). The third item, a traceback object, is new. This object provides the execution context of where the exception occurred. It contains information such as the execution frame of the code that was running and the line number where the exception occurred.

In older versions of Python, these three values were available in the `sys` module as `sys.exc_type`, `sys.exc_value`, and `sys.exc_traceback`. Unfortunately, these three are global variables and not thread-safe. We recommend using `sys.exc_info()` instead. All three will be phased out and eventually removed in a future version of Python.

10.13. Related Modules

Table 10.3 lists some of the modules related to this chapter.

Table 10.3. Exception-Related Standard Library Modules

Module	*Description*
`exceptions`	Built-in exceptions (never need to import this module)
`contextlib`[a]	Context object utilities for use **`with`** the with statement
`sys`	Contains various exception-related objects and functions (see `sys.ex*`)

[a] New in Python 2.5.

10.14. Exercises

10-1. *Raising Exceptions*. Which of the following can *raise* exceptions during program execution? Note that this question does not ask what may *cause* exceptions.

a.

The user (of your program)

b.

The interpreter

c.

The program(er)

d.

All of the above

e.

Only (b) and (c)

f.

Only (a) and (c)

10-2. *Raising Exceptions.* Referring to the list in the problem above, which could raise exceptions while running within the interactive interpreter?

10-3. *Keywords*. Name the keyword(s) which is (are) used to raise exceptions.

10-4. *Keywords.* What is the difference between `try-except` and `TRy-finally`?

10-5. *Exceptions*. Name the exception that would result from executing the following pieces of Python code from within the interactive interpreter (refer back to Table 10.2 for a list of all built-in exceptions):

```
(a)  >>> if 3 < 4 then: print '3 IS less than 4!'

(b)  >>> aList = ['Hello', 'World!', 'Anyone',
'Home?']
     >>> print 'the last string in aList is:', aList
     [len(aList)]

(c)  >>> x

(d)  >>> x = 4 % 0

(e)  >>> import math
    >>> i = math.sqrt(-1)
```

10-6. *Improving open()*. Create a wrapper for the `open()` function. When a program opens a file successfully, a file handle will be returned. If the file open fails, rather than generating an error, return `None` to the callers so that they can open files without an exception handler.

10-7. *Exceptions*. What is the difference between Python pseudocode snippets (a) and (b)? Answer in the context of statements A and B, which are part of both pieces of code. (Thanks to Guido for this teaser!)

```
(a)  try:
        statement_A
   except . . .:
        . . .
     else:
        statement_B

(b)  try:
        statement_A
        statement_B
    except . . .:
          . . .
```

10-8. *Improving `raw_input()`*. At the beginning of this chapter, we presented a "safe" version of the `float()` built-in function to detect and handle two different types of exceptions that `float()` generates. Likewise, the `raw_input()` function can generate two different exceptions, either `EOFError` or `KeyboardInterrupt` on end-of-file (EOF) or cancelled input, respectively. Create a wrapper function, perhaps `safe_ input()`; rather than raising an exception if the user entered EOF (^D in Unix or ^Z in DOS) or attempted to break out using ^C, have your function return `None` that the calling function can check for.

10-9. *Improving* `math.sqrt()`. The `math` module contains many functions and some constants for performing various mathematics-related operations. Unfortunately, this module does not recognize or operate on complex numbers, which is the reason why the `cmath` module was developed. Create a function, perhaps `safe_sqrt()`, which wraps `math.sqrt()`, but is smart enough to handle a negative parameter and return a complex number with the correct value back to the caller.

PREV NEXT

Chapter 12. Modules

Chapter Topics

This chapter focuses on Python modules and how data are imported from modules into your programming environment. We will also take a look at packages. Modules are a means to organize Python code, and packages help you organize modules. We conclude this chapter with a look at other related aspects of modules.

12.1. What Are Modules?

A *module* allows you to logically organize your Python code. When code gets to be large enough, the tendency is to break it up into organized pieces that can still interact with one another at a functioning level. These pieces generally have attributes that have some relation to one another, perhaps a single class with its member data variables and methods, or maybe a group of related, yet independently operating functions. These pieces should be shared, so Python allows a module the ability to "bring in" and use attributes from other modules to take advantage of work that has been done, maximizing code reusability. This process of associating attributes from other modules with your module is called *importing*. In a nutshell, modules are self-contained and organized pieces of Python code that can be shared.

12.2. Modules and Files

If modules represent a logical way to organize your Python code, then files are a way to physically organize modules. To that end, each file is considered an individual module, and vice versa. The filename of a module is the module name appended with the `.py` file extension. There are several aspects we need to discuss with regard to what the file structure means to modules. Unlike other languages in which you import classes, in Python you import modules or module attributes.

12.2.1. Module Namespaces

We will discuss namespaces in detail later in this chapter, but the basic concept of a namespace is an individual set of mappings from names to objects. As you are no doubt aware, module names play an important part in the naming of their attributes. The name of the attribute is always prepended with the module name. For example, the `atoi()` function in the `string` module is called `string.atoi()`. Because only one module with a given name can be loaded into the Python interpreter, there is no intersection of names from different modules; hence, each module defines its own unique namespace. If I created a function called `atoi()` in my own module, perhaps `mymodule`, its name would be `mymodule.atoi()`. So even if there is a name conflict for an attribute, the *fully qualified name*referring to an object via dotted attribute notationprevents an exact and conflicting match.

12.2.2. Search Path and Path Search

The process of importing a module requires a process called a *path search*. This is the procedure of checking "predefined areas" of the file system to look for your `mymodule.py` file in order to load the `mymodule` module. These predefined areas are no more than a set of directories that are part of your Python *search path*. To avoid the confusion between the two, think of a path search as the pursuit of a file through a set of directories, the search path.

There may be times where importing a module fails:

```
>>> import xxx
Traceback (innermost last):
 File "<interactive input>", line 1, in ?
ImportError: No module named xxx
```

When this error occurs, the interpreter is telling you it cannot access the requested module, and the likely reason is that the module you desire is not in the search path, leading to a path search failure.

A default search path is automatically defined either in the compilation or installation process. This search path may be modified in one of two places.

One is the `PYTHONPATH` environment variable set in the *shell* or command-line interpreter that invokes Python. The contents of this variable consist of a colon-delimited set of directory paths. If you want the interpreter to use the contents of this variable, make sure you set or update it before you start the interpreter or run a Python script.

Once the interpreter has started, you can access the path itself, which is stored in the `sys` module as the `sys.path` variable. Rather than a single string that is colon-delimited, the path has been "split" into a list of individual directory strings. Below is an example search path for a Unix machine. Your mileage will

definitely vary as you go from system to system.

```
>>> sys.path
['', '/usr/local/lib/python2.x/', '/usr/local/lib/
python2.x/plat-sunos5', '/usr/local/lib/python2.x/
lib-tk', '/usr/local/lib/python2.x/lib-dynload', '/
usr/local/lib/Python2.x/site-packages',]
```

Bearing in mind that this is just a list, we can definitely take liberty with it and modify it at our leisure. If you know of a module you want to import, yet its directory is not in the search path, by all means use the list's `append()` method to add it to the path, like so:

```
sys.path.append('/home/wesc/py/lib')
```

Once this is accomplished, you can then load your module. As long as one of the directories in the search path contains the file, then it will be imported. Of course, this adds the directory only to the end of your search path. If you want to add it elsewhere, such as in the beginning or middle, then you have to use the `insert()` list method for those. In our examples above, we are updating the `sys.path` attribute interactively, but it will work the same way if run as a script.

Here is what it would look like if we ran into this problem interactively:

```
>>> import sys
>>> import mymodule
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named mymodule
>>>
>>> sys.path.append('/home/wesc/py/lib')
>>> sys.path
['', '/usr/local/lib/python2.x/', '/usr/local/lib/
python2.x/plat-sunos5', '/usr/local/lib/python2.x/
lib-tk', '/usr/local/lib/python2.x/lib-dynload', '/usr/
local/lib/python2.x/site-packages','/home/wesc/py/lib']
>>>
>>> import mymodule
>>>
```

On the flip side, you may have too many copies of a module. In the case of duplicates, the interpreter will load the first module it finds with the given name while rummaging through the search path in sequential order.

To find out what modules have been successfully imported (and loaded) as well as from where, take a look at `sys.modules`. Unlike `sys.path`, which is a list of modules, `sys.modules` is a dictionary where the keys are the module names with their physical location as the values.

12.3. Namespaces

A *namespace* is a mapping of names (identifiers) to objects. The process of adding a name to a namespace consists of *binding* the identifier to the object (and increasing the reference count to the object by one). The Python Language Reference also includes the following definitions: "changing the mapping of a name is called *rebinding* [, and] removing a name is *unbinding*."

As briefly introduced in Chapter 11, there are either two or three active namespaces at any given time during execution. These three namespaces are the local, global, and built-ins namespaces, but local name-spaces come and go during execution, hence the "two or three" we just alluded to. The names accessible from these namespaces are dependent on their *loading order*, or the order in which the namespaces are brought into the system.

The Python interpreter loads the built-ins namespace first. This consists of the names in the `__builtins__` module. Then the global namespace for the executing module is loaded, which then becomes the active namespace when the module begins execution. Thus we have our two active namespaces.

Core Note: `__builtins__` versus `__builtin__`

The `__builtins__` module should not be confused with the `__builtin__` module. The names, of course, are so similar that it tends to lead to some confusion among new Python programmers who have gotten this far. The `__builtins__` module consists of a set of built-in names for the built-ins namespace. Most, if not all, of these names come from the `__builtin__` module, which is a module of the built-in functions, exceptions, and other attributes. In standard Python execution, `__builtins__` contains all the names from `__builtin__`. Python used to have a restricted execution model that allowed modification of `__builtins__` where key pieces from `__builtin__` were left out to create a sandbox environment. However, due its security flaws and the difficulty involved with repairing it, restricted execution is no longer supported in Python (as of 2.3).

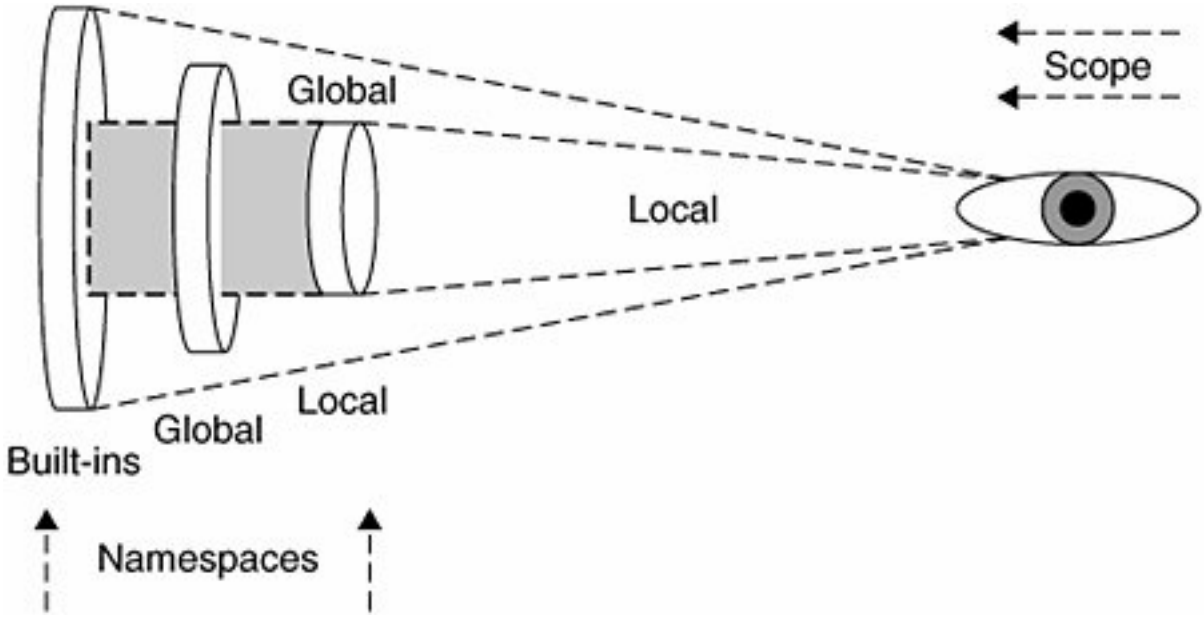
When a function call is made during execution, the third, a local, namespace is created. We can use the `globals()` and `locals()` built-in functions to tell us which names are in which namespaces. We will discuss both functions in more detail later on in this chapter.

12.3.1. Namespaces versus Variable Scope

Okay, now that we know what namespaces are, how do they relate to variable scope again? They seem extremely similar. The truth is, you are quite correct.

Namespaces are purely mappings between names and objects, but scope dictates how, or rather where, one can access these names based on the physical location from within your code. We illustrate the relationship between namespaces and variable scope in Figure 12-1.

Figure 12-1. Namespaces versus variable scope



Notice that each of the namespaces is a self-contained unit. But looking at the namespaces from the scoping point of view, things appear different. All names within the local namespace are within my local scope. Any name outside my local scope is in my global scope.

Also keep in mind that during the execution of the program, the local namespaces and scope are transient because function calls come and go, but the global and built-ins namespaces remain.

Our final thought to you in this section is, when it comes to namespaces, ask yourself the question, "Does it have it?" And for variable scope, ask, "Can I see it?"

12.3.2. Name Lookup, Scoping, and Overriding

So how do scoping rules work in relationship to namespaces? It all has to do with name lookup. When accessing an attribute, the interpreter must find it in one of the three namespaces. The search begins with the local namespace. If the attribute is not found there, then the global namespace is searched. If that is also unsuccessful, the final frontier is the built-ins namespace. If the exhaustive search fails, you get the familiar:

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

Notice how the figure features the foremost-searched namespaces "shadowing" namespaces, which are searched afterward. This is to try to convey the effect of *overriding*. This shadowing effect is illustrated by the gray boxes in Figure 12-1. For example, names found in the local namespace will hide access to objects in the global or built-ins namespaces. This is the process whereby names may be taken out of scope because a more local namespace contains a name. Take a look at the following piece of code that was introduced in the previous chapter:

```
def foo():
    print "\ncalling foo()..."
    bar = 200
    print "in foo(), bar is", bar
```

```
bar = 100
print "in __main__, bar is", bar
foo()
```

When we execute this code, we get the following output:

```
in __main__, bar is 100

calling foo()...
in foo(), bar is 200
```

The `bar` variable in the local namespace of `foo()` overrode the global `bar` variable. Although `bar` exists in the global namespace, the lookup found the one in the local namespace first, hence "overriding" the global one. For more information regarding scope, see Section 11.8 of Chapter 11.

12.3.3. Namespaces for Free!

One of Python's most useful features is the ability to get a namespace almost anywhere you need a place to put things. We have seen in the previous chapter how you can just add attributes to functions at whim (using the familiar dotted-attribute notation):

```
def foo():
     pass
foo.__doc__ = 'Oops, forgot to add doc str above!'
foo.version = 0.2
```

In this chapter, we have shown how modules themselves make namespaces and how you access them in the same way:

```
mymodule.foo()
mymodule.version
```

Although we will discuss object-oriented programming (OOP) in Chapter 13, how about an example even simpler than a "Hello World!" to introduce you to Python classes?

```
class MyUltimatePythonStorageDevice(object):
     pass

bag = MyUltimatePythonStorageDevice()
bag.x = 100
bag.y = 200
bag.version = 0.1
bag.completed = False
```

You can throw just about anything you want in a namespace. This use of a class (instance) is perfectly fine, and you don't even have to know much about OOP to be able to use a class! (Note: These guys are called *instance attributes*.) Fancy names aside, the instance is just used as a namespace.

You will see just how useful they are as you delve deeper into OOP and discover what a convenience it is during runtime just to be able to store temporary (but important) values! As stated in the final tenet of the Zen of Python:

"Namespaces are one honking great idealet's do more of those!"

(To see the complete Zen, just import the `this` module within the interactive interpreter.)

PREV NEXT

12.4. Importing Modules

12.4.1. The `import` Statement

Importing a module requires the use of the **`import`** statement, whose syntax is:

```
import module1

import module2[
    :
import moduleN
```

It is also possible to import multiple modules on the same line like this ...

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

... but the resulting code is not as readable as having multiple import statements. Also, there is no performance hit and no change in the way that the Python bytecode is generated, so by all means, use the first form, which is the preferred form.

Core Style: Module ordering for `import` statements

It is recommended that all module imports happen at the top of Python modules. Furthermore, imports should follow this ordering:

- *Python Standard Library modules*
- *Python third party modules*
- *Application-specific modules*

Separate these groups with an empty line between the imports of these three types of modules. This helps ensure that modules are imported in a consistent manner and helps minimize the number of ***`import`*** *statements required in each of the modules. You can read more about this and other import tips in Python's Style Guide, written up as PEP 8.*

When this statement is encountered by the interpreter, the module is imported if found in the search path. Scoping rules apply, so if imported from the top level of a module, it has global scope; if imported from a function, it has local scope.

When a module is imported the first time, it is loaded and executed.

12.4.2. The `from-import` Statement

It is possible to import specific module elements into your own module. By this, we really mean importing specific names from the module into the current namespace. For this purpose, we can use the **from-import** statement, whose syntax is:

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

12.4.3. Multi-Line Import

The multi-line import feature was added in Python 2.4 specifically for long **from-import** statements. When importing many attributes from the same module, import lines of code tend to get long and wrap, requiring a NEWLINE-escaping backslash. Here is the example *imported* (pun intended) directly from PEP 328:

2.4

```
from Tkinter import Tk, Frame, Button, Entry, Canvas, \
            Text, LEFT, DISABLED, NORMAL, RIDGE, END
```

Your other option is to have multiple **from-import** statements:

```
from Tkinter import Tk, Frame, Button, Entry, Canvas, Text
from Tkinter import LEFT, DISABLED, NORMAL, RIDGE, END
```

We are also trying to stem usage on the unfavored **from** `Tkinter` **import** `*` (see the Core Style sidebar in Section 12.5.3). Instead, programmers should be free to use Python's standard grouping mechanism (parentheses) to create a more reasonable multi-line **import** statement:

```
from Tkinter import (Tk, Frame, Button, Entry, Canvas,
            Text, LEFT, DISABLED, NORMAL, RIDGE, END)
```

You can find out more about multi-line imports in the documentation or in PEP 328.

12.4.4. Extended **Import** Statement (**as**)

There are times when you are importing either a module or module attribute with a name that you are already using in your application, or perhaps it is a name that you do not want to use. Maybe the name is too long to type everywhere, or more subjectively, perhaps it is a name that you just plain do not like.

2.0

This had been a fairly common request from Python programmers: the ability to import modules and module attributes into a program using names other than their original given names. One common

workaround is to assign the module name to a variable:

```
>>> import longmodulename
>>> short = longmodulename
>>> del longmodulename
```

In the example above, rather than using `longmodulename.attribute`, you would use the `short.attribute` to access the same object. (A similar analogy can be made with importing module attributes using **`from-import`**, see below.) However, to do this over and over again and in multiple modules can be annoying and seem wasteful. Using extended import, you can change the locally bound name for what you are importing. Statements like ...

```
    import Tkinter
from cgi import FieldStorage
```

. . . can be replaced by . . .

```
import Tkinter as tk
from cgi import FieldStorage as form
```

2.0-2.6

This feature was added in Python 2.0. At that time, "`as`" was not implemented as a keyword; it finally became one in Python 2.6. For more information on extended import, see the Python Language Reference Manual and PEP 221.

12.5. Features of Module Import

12.5.1. Module "Executed" When Loaded

One effect of loading a module is that the imported module is "executed," that is, the top-level portion of the imported module is directly executed. This usually includes setting up of global variables as well as performing the class and function declarations. If there is a check for `__name__` to do more on direct script invocation, that is executed, too.

Of course, this type of execution may or may not be the desired effect. If not, you will have to put as much code as possible into functions. Suffice it to say that good module programming style dictates that only function and/or class definitions should be at the top level of a module.

For more information see Section 14.1.1 and the Core Note contained therein.

A new feature was added to Python which allows you to execute an installed module as a script. (Sure, running your own script is easy [`$ foo.py`], but executing a module in the standard library or third party package is trickier.) You can read more about how to do this in Section 14.4.3.

12.5.2. Importing versus Loading

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. If your module imports the `sys` module, and so do five of the other modules you import, it would not be wise to load `sys` (or any other module) each time! So rest assured, loading happens only once, on first import.

12.5.3. Names Imported into Current Namespace

Calling **`from-import`** brings the name into the current namespace, meaning that you do not use the attribute/dotted notation to access the module identifier. For example, to access a variable named *`var`* in module *`module`* that was imported with:

```
from module import var
```

we would use "*`var`*" by itself. There is no need to reference the module since you imported *`var`* into your namespace. It is also possible to import all the names from the module into the current namespace using the following **`from-import`** statement:

```
from module import *
```

Core Style: Restrict your use of "`from module import *`"

In practice, using **`from`** *`module`* **`import`** *`*`* *is considered poor style because it "pollutes" the current namespace and has the potential of overriding names in the current namespace; however, it is extremely convenient if a module has many variables that are often accessed, or if the module has a very long name.*

We recommend using this form in only two situations. The first is where the target module has many attributes that would make it inconvenient to type in the module name over and over again. Two prime examples of this are the `Tkinter` (Python/Tk) and `NumPy` (Numeric Python) modules, and perhaps the `socket` module. The other place where it is acceptable to use **`from`** *`module`* **`import`** *`*`* *is within the interactive interpreter, to save on the amount of typing.*

12.5.4. Names Imported into Importer's Scope

Another side effect of importing just names from modules is that those names are now part of the local namespace. A side effect is possibly hiding or overriding an existing object or built-in with the same name. Also, changes to the variable affect only the local copy and not the original in the imported module's namespace. In other words, the binding is now local rather than across namespaces.

Here we present the code to two modules: an importer, `impter.py`, and an importee, `imptee.py`. Currently, `imptr.py` uses the **`from-import`** statement, which creates only local bindings.

```
#############
# imptee.py #
#############
foo = 'abc'
def show():
    print 'foo from imptee:', foo

#############
# impter.py #
#############
from imptee import foo, show
show()
foo = 123
print 'foo from impter:', foo
show()
```

Upon running the importer, we discover that the importee's view of its `foo` variable has not changed even though we modified it in the importer.

```
foo from imptee: abc
foo from impter: 123
foo from imptee: abc
```

The only solution is to use import and *fully qualified* identifier names using the attribute/dotted notation.

```
#############
```

```
# impter.py #
#############
import imptee
imptee.show()
imptee.foo = 123
print 'foo from impter:', imptee.foo
imptee.show()
```

Once we make the update and change our references accordingly, we now have achieved the desired effect.

```
foo from imptee: abc
foo from impter: 123
foo from imptee: 123
```

12.5.5. Back to the `__future__`

Back in the days of Python 2.0, it was recognized that due to improvements, new features, and current feature enhancements, certain significant changes could not be implemented without affecting some existing functionality. To better prepare Python programmers for what was coming down the line, the `__future__` directives were implemented.

By using the **`from-import`** statement and "importing" future functionality, users can get a taste of new features or feature changes enabling them to port their applications correctly by the time the feature becomes permanent. The syntax is:

```
from __future__ import new_feature
```

It does not make sense to import `__future__` so that is disallowed. (Actually, it is allowed but does not do what you want it to do, which is enable all future features.) You have to import specific features explicitly. You can read more about `__future__` directives in PEP 236.

12.5.6. Warning Framework

Similar to the `__future__` directive, it is also necessary to warn users when a feature is about to be changed or deprecated so that they can take action based on the notice received. There are multiple pieces to this feature, so we will break it down into components.

2.1

The first piece is the application programmer's interface (API). Programmers have the ability to issue warnings from both Python programs (via the `warnings` module) as well as from C [via a call to `PyErr_Warn()`].

Another part of the framework is a new set of warning exception classes. `Warning` is subclassed directly from `Exception` and serves as the root of all warnings: `UserWarning`, `DeprecationWarning`, `SyntaxWarning`,

and `RuntimeWarning`. These are described in further detail in Chapter 10.

The next component is the warnings filter. There are different warnings of different levels and severities, and somehow the number and type of warnings should be controllable. The warnings filter not only collects information about the warning, such as line number, cause of the warning, etc., but it also controls whether warnings are ignored, displayedthey can be custom-formattedor turned into errors (generating an exception).

Warnings have a default output to `sys.stderr`, but there are hooks to be able to change that, for example, to log it instead of displaying it to the end-user while running Python scripts subject to issued warnings. There is also an API to manipulate warning filters.

Finally, there are the command-line arguments that control the warning filters. These come in the form of options to the Python interpreter upon startup via the `-W` option. See the Python documentation or PEP 230 for the specific switches for your version of Python. The warning framework first appeared in Python 2.1.

12.5.7. Importing Modules from ZIP Files

In version 2.3, the feature that allows the import of modules contained inside ZIP archives was added to Python. If you add a `.zip` file containing Python modules (`.py, .pyc`, or `.pyo` files) to your search path, i.e., `PYTHONPATH` or `sys.path`, the importer will search that archive for the module as if the ZIP file was a directory.

2.3

If a ZIP file contains just a `.py` for any imported module, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive does not contain a matching `.pyc` file, import speed should be expected to be slower than if they were present.

You are also allowed to add specific (sub)directories "under" a `.zip` file, i.e., `/tmp/yolk.zip/lib/` would only import from the `lib/` subdirectory within the `yolk` archive. Although this feature is specified in PEP 273, the actual implementation uses the import hooks provided by PEP 302.

12.5.8. "New" Import Hooks

The import of modules inside ZIP archives was "the first customer" of the new import hooks specified by PEP 302. Although we use the word "new," that is relative considering that it has been difficult to create custom importers because the only way to accomplish this before was to use the other modules that were either really old or didn't simplify writing importers. Another solution is to override `__import__()`, but that is not an easy thing to do because you have to pretty much (re)implement the entire import mechanism.

2.3

The new import hooks, introduced in Python 2.3, simplify it down to writing callable import classes, and

getting them "registered" (or rather, "installed") with the Python interpreter via the `sys` module.

There are two classes that you need: a finder and a loader. An instance of these classes takes an argumentthe full name of any module or package. A finder instance will look for your module, and if it finds it, return a loader object. The finder can also take a path for finding subpackages. The loader is what eventually brings the module into memory, doing whatever it needs to do to make a real Python module object, which is eventually returned by the loader.

These instances are added to `sys.path_hooks`. The `sys.path_importer_ cache` just holds the instances so that `path_hooks` is traversed only once. Finally, `sys.meta_path` is a list of instances that should be traversed before looking at `sys.path`, for modules whose location you know and do not need to find. The meta-path already has the loader objects reader to execute for specific modules or packages.

PREV NEXT

12.6. Module Built-in Functions

The importation of modules has some functional support from the system. We will look at those now.

12.6.1. `__import__ ()`

The `__import__()` function is new as of Python 1.5, and it is the function that actually does the importing, meaning that the import statement invokes the `__import__()` function to do its work. The purpose of making this a function is to allow for overriding it if the user is inclined to develop his or her own importation algorithm.

The syntax of `__import__()` is:

```
__import__(module_name[, globals[, locals[, fromlist]]])
```

The *module_name* variable is the name of the module to import, *globals* is the dictionary of current names in the global symbol table, *locals* is the dictionary of current names in the local symbol table, and *fromlist* is a list of symbols to import the way they would be imported using the **from-import** statement.

The *globals*, *locals*, and *fromlist* arguments are optional, and if not provided, default to `globals()`, `locals()`, and `[]`, respectively.

Calling **import** `sys` can be accomplished with

```
sys = __import__('sys')
```

12.6.2. `globals()` and `locals()`

The `globals()` and `locals()` built-in functions return dictionaries of the global and local namespaces, respectively, of the caller. From within a function, the local namespace represents all names defined for execution of that function, which is what `locals()` will return. `globals()`, of course, will return those names globally accessible to that function.

From the global namespace, however, `globals()` and `locals()` return the same dictionary because the global namespace is as local as you can get while executing there. Here is a little snippet of code that calls both functions from both namespaces:

```
def foo():
    print '\ncalling foo()...'
    aString = 'bar'
    anInt = 42
    print "foo()'s globals:", globals().keys()
    print "foo()'s locals:", locals().keys()

print "__main__'s globals:", globals().keys()
print "__main__'s locals:", locals().keys()
```

```
foo()
```

We are going to ask for the dictionary keys only because the values are of no consequence here (plus they make the lines wrap even more in this text). Executing this script, we get the following output:

```
$ namespaces.py
__main__'s globals: ['__doc__', 'foo', '__name__',
'__builtins__']
__main__'s locals: ['__doc__', 'foo', '__name__',
'__builtins__']

calling foo()...
foo()'s globals: ['__doc__', 'foo', '__name__',
'__builtins__']
foo()'s locals: ['anInt', 'aString']
```

12.6.3. `reload()`

The `reload()` built-in function performs another import on a previously imported module. The syntax of `reload()` is:

```
reload(module)
```

`module` is the actual module you want to reload. There are some criteria for using the `reload()` module. The first is that the module must have been imported in full (not by using **`from-import`**), and it must have loaded successfully. The second rule follows from the first, and that is the argument to `reload()` the module itself and not a string containing the module name, i.e., it must be something like `reload(sys)` instead of `reload('sys')`.

Also, code in a module is executed when it is imported, but only once. A second import does not re-execute the code, it just binds the module name. Thus `reload()` makes sense, as it overrides this default behavior.

12.7. Packages

A *package* is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages. Packages were added to Python 1.5 to aid with a variety of problems including:

- Adding hierarchical organization to flat namespace
- Allowing developers to group related modules
- Allowing distributors to ship directories vs. bunch of files
- Helping resolve conflicting module names

Along with classes and modules, packages use the familiar attribute/dotted attribute notation to access their elements. Importing modules within packages use the standard **`import`** and **`from-import`** statements.

12.7.1. Directory Structure

For our package examples, we will assume the directory structure below:

```
Phone/
    __init__.py
    common_util.py
    Voicedta/
        __init__.py
        Pots.py
        Isdn.py
    Fax/
        __init__.py
        G3.py
    Mobile/
        __init__.py
        Analog.py
        Digital.py
    Pager/
         __init__.py
         Numeric.py
```

`Phone` is a top-level package and `Voicedta`, etc., are subpackages. Import subpackages by using **`import`** like this:

```
import Phone.Mobile.Analog
Phone.Mobile.Analog.dial()
```

Alternatively, you can use **`from-import`** in a variety of ways:

The first way is importing just the top-level subpackage and referencing down the subpackage tree using the attribute/dotted notation:

```
from Phone import Mobile
Mobile.Analog.dial('555-1212')
```

Furthermore, we can go down one more subpackage for referencing:

```
from Phone.Mobile import Analog
Analog.dial('555-1212')
```

In fact, you can go all the way down in the subpackage tree structure:

```
from Phone.Mobile.Analog import dial
dial('555-1212')
```

In our above directory structure hierarchy, we observe a number of `__init__.py` files. These are initializer modules that are required when using **`from-import`** to import subpackages but they can be empty if not used. Quite often, developers forget to add `_inti_.py` files to their package directories, so starting in Python 2.5, this triggers an `ImportWarning` message.

However, it is silently ignored unless the `-Wd` option is given when launching the interpreter.

12.7.2. Using `from-import` with Packages

Packages also support the **`from-import`** all statement:

```
from package.module import *
```

However, such a statement is dependent on the operating system's filesystem for Python to determine which files to import. Thus the `__all__` variable in `__init__.py` is required. This variable contains all the module names that should be imported when the above statement is invoked if there is such a thing. It consists of a list of module names as strings.

12.7.3. Absolute Import

As the use of packages becomes more pervasive, there have been more cases of the import of subpackages that end up clashing with (and hiding or shadowing) "real" or standard library modules (actually their names). Package modules will hide any equivalently-named standard library module because it will look inside the package first to perform a *relative import*, thus hiding access to the standard library module.

Because of this, all imports are now classified as *absolute*, meaning that names must be packages or modules accessible via the Python path (`sys.path` or `PYTHONPATH`).

2.5-2.7

The rationale behind this decision is that subpackages can still be accessed via `sys.path`, i.e., `import Phone.Mobile.Analog`. Prior to this change, it was legal to have just `import Analog` from modules inside the `Mobile` subpackage.

As a compromise, Python allows relative importing where programmers can indicate the location of a subpackage to be imported by using leader dots in front of the module or package name. For more information, please see Section 12.7.4.

The absolute import feature is the default starting in Python 2.7. (This feature, `absolute_import`, can be imported from `__future__` starting in version 2.5.) You can read more about absolute import in PEP 328.

12.7.4. Relative Import

As described previously, the absolute import feature takes away certain privileges of the module writer of packages. With this loss of freedom in `import` statements, something must be made available to proxy for that loss. This is where a relative import comes in. The relative import feature alters the import syntax slightly to let programmers tell the importer where to find a module in a subpackage. Because the `import` statements are always absolute, relative imports only apply to `from-import` statements.

2.5-2.6

The first part of the syntax is a leader dot to indicate a relative import. From there, any additional dot represents a single level above the current from where to start looking for the modules being imported.

Let us look at our example above again. From within `Analog.Mobile. Digital`, i.e., the `Digital.py` module, we cannot simply use this syntax anymore. The following will either still work in older versions of Python, generate a warning, or will not work in more contemporary versions of Python:

```
import Analog
from Analog import dial
```

This is due to the absolute import limitation. You have to use either the absolute or relative imports. Below are some valid imports:

```
from Phone.Mobile.Analog import dial
from .Analog import dial
from ..common_util import setup
from ..Fax import G3.dial.
```

Relative imports can be used starting in Python 2.5. In Python 2.6, a deprecation warning will appear for all intra-package imports not using the relative import syntax. You can read more about relative import in the Python documentation and in PEP 328.

12.8. Other Features of Modules

12.8.1. Auto-Loaded Modules

When the Python interpreter starts up in standard mode, some modules are loaded by the interpreter for system use. The only one that affects you is the `__builtin__` module, which normally gets loaded in as the `__builtins__` module.

The `sys.modules` variable consists of a dictionary of modules that the interpreter has currently loaded (in full and successfully) into the interpreter. The module names are the keys, and the location from which they were imported are the values.

For example, in Windows, the `sys.modules` variable contains a large number of loaded modules, so we will shorten the list by requesting only the module names. This is accomplished by using the dictionary's `keys()` method:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'exceptions', '__main__', 'ntpath',
'strop', 'nt', 'sys',  '__builtin__', 'site',
'signal', 'UserDict', 'string', 'stat']
```

The loaded modules for Unix are quite similar:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'readline', 'exceptions',
'__main__', 'posix', 'sys', '__builtin__', 'site',
'signal', 'UserDict', 'posixpath', 'stat']
```

12.8.2. Preventing Attribute Import

If you do not want module attributes imported when a module is imported with "**from** *module* **import** *", prepend an underscore (_) to those attribute names (you do not want imported). This minimal level of data hiding does not apply if the entire module is imported or if you explicitly import a "hidden" attribute, e.g., **import** foo._bar.

12.8.3. Case-Insensitive Import

There are various operating systems with case-insensitive file systems. Prior to version 2.1, Python attempted to "do the right thing" when importing modules on the various supported platforms, but with the growing popularity of the MacOS X and Cygwin platforms, certain deficiencies could no longer be ignored, and support needed to be cleaned up.

The world was pretty clean-cut when it was just Unix (case-sensitive) and Win32 (case-insensitive), but these new case-insensitive systems coming online were not ported with the case-insensitive features. PEP 235, which specifies this feature, attempts to address this weakness as well as taking away some "hacks" that had existed for other systems to make importing modules more consistent.

The bottom line is that for case-insensitive imports to work properly, an environment variable named `PYTHONCASEOK` must be defined. Python will then import the first module name that is found (in a case-insensitive manner) that matches. Otherwise Python will perform its native case-sensitive module name matching and import the first matching one it finds.

12.8.4. Source Code Encoding

Starting in Python 2.3, it is now possible to create your Python module file in a native encoding other than 7-bit ASCII. Of course ASCII is the default, but with an additional encoding directive at the top of your Python modules, it will enable the importer to parse your modules using the specified encoding and designate natively encoded Unicode strings correctly so you do not have to worry about editing your source files in a plain ASCII text editor and have to individually "Unicode-tag" each string literal.

An example directive specifying a UTF-8 file can be declared like this:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

If you execute or import modules that contain non-ASCII Unicode string literals and do not have an encoding directive at the top, this will result in a `DeprecationWarning` in Python 2.3 and a syntax error starting in 2.5. You can read more about source code encoding in PEP 263.

12.8.5. Import Cycles

Working with Python in real-life situations, you discover that it is possible to have import loops. If you have ever worked on any large Python project, you are likely to have run into this situation.

Let us take a look at an example. Assume we have a very large product with a very complex command-line interface (CLI). There are a million commands for your product, and as a result, you have an overly massive handler (OMH) set. Every time a new feature is added, from one to three new commands must be added to support the new feature. This will be our `omh4cli.py` script:

```
from cli4vof import cli4vof

# command line interface utility function
def cli_util():
    pass
```

```
# overly massive handlers for the command line interface
def omh4cli():
        :
    cli4vof()
        :
omh4cli()
```

You can pretend that the (empty) utility function is a very popular piece of code that most handlers must use. The overly massive handlers for the command-line interface are all in the `omh4cli()` function. If we have to add a new command, it would be called from here.

Now, as this module grows in a boundless fashion, certain smarter engineers decide to split off their new commands into a separate module and just provide hooks in the original module to access the new stuff. Therefore, the code is easier to maintain, and if bugs were found in the new stuff, one would not have to search through a one-megabyte-plus-sized Python file.

In our case, we have an excited product manager asking us to add a "very outstanding feature" (VOF). Instead of integrating our stuff into `omh4cli.py`, we create a new script, `cli4vof.py`:

```
import omh4cli

# command-line interface for a very outstanding feature
def cli4vof():
    omh4cli.cli_util()
```

As mentioned before, the utility function is a must for every command, and because we do not want to cut and paste its code from the main handler, we import the main module and call it that way. To finish off our integration, we add a call to our handler into the main overly massive handler, `omh4cli()`.

The problem occurs when the main handler `omh4cli` imports our new little module `cli4vof` (to get the new command function) because `cli4vof` imports `omh4cli` (to get the utility function). Our module import fails because Python is trying to import a module that was not previously fully imported the first time:

```
$ python omh4cli.py
Traceback (most recent call last):
  File "omh4cli.py", line 3, in ?
    from cli4vof import cli4vof
  File "/usr/prod/cli4vof.py", line 3, in ?
    import omh4cli
  File "/usr/prod/omh4cli.py", line 3, in ?
    from cli4vof import cli4vof
ImportError: cannot import name cli4vof
```

Notice the circular import of `cli4vof` in the traceback. The problem is that in order to call the utility function, `cli4vof` has to import `omh4cli`. If it did not have to do that, then `omh4cli` would have completed its import of `cli4vof` successfully and there would be no problem. The issue is that when `omh4cli` is attempting to import `cli4vof`, `cli4vof` is trying to import `omh4cli`. No one finishes an import, hence the error. This is just one example of an import cycle. There are much more complicated ones out in the real world.

The workaround for this problem is almost always to move one of the **import** statements, e.g., the offending one. You will commonly see **import** statements at the bottom of modules. As a beginning Python programmer, you are used to seeing them in the beginning, but if you ever run across **import** statements at the end of modules, you will now know why. In our case, we cannot move the import of omh4cli to the end, because if cli4vof() is called, it will not have the omh4cli name loaded yet:

```
$ python omh4cli.py
Traceback (most recent call last):
  File "omh4cli.py", line 3, in ?
    from cli4vof import cli4vof
  File "/usr/prod/cli4vof.py", line 7, in ?
    import omh4cli
  File "/usr/prod/omh4cli.py", line 13, in ?
    omh4cli()
  File "/usr/prod/omh4cli.py", line 11, in omh4cli
    cli4vof()
  File "/usr/prod/cli4vof.py", line 5, in cli4vof
    omh4cli.cli_util()
NameError: global name 'omh4cli' is not defined
```

No, our solution here is to just move the **import** statement into the cli4vof() function declaration:

```
def cli4vof():
    import omh4cli
    omh4cli.cli_util()
```

This way, the import of the cli4vof module from omh4cli completes successfully, and on the tail end, calling the utility function is successful because the omh4cli name is imported before it is called. As far as execution goes, the only difference is that from cli4vof, the import of omh4cli is performed when cli4vof.cli4vof() is called and not when the cli4vof module is imported.

12.8.6. Module Execution

There are many ways to execute a Python module: script invocation via the command-line or shell, execfile(), module import, interpreter -m option, etc. These are out of the scope of this chapter. We refer you to Chapter 14, "Execution Environment," which covers all of these features in full detail.

12.9. Related Modules

The following are auxiliary modules that you may use when dealing with the import of Python modules. Of these listed below, `modulefinder`, `pkgutil`, and `zipimport` are new as of Python 2.3, and the `distutils` package was introduced back in version 2.0.

2.0-2.3

- `imp` this module gives you access to some lower-level importer functionality.
- `modulefinder` this is a module that lets you find all the modules that are used by a Python script. You can either use the `ModuleFinder` class or just run it as a script giving it the filename of a (nother) Python module with which to do module analysis on.
- `pkgutil` this module gives those putting together Python packages for distribution a way to place package files in various places yet maintain the abstraction of a single "package" file hierarchy. It uses `*.pkg` files in a manner similar to the way the site module uses `*.pth` files to help define the package path.
- `site` using this module along with `*.pth` files gives you the ability to specify the order in which packages are added to your Python path, i.e., `sys.path`, `PYTHONPATH`. You do not have to import it explicitly as the importer already uses it by defaultyou need to use the `-S` switch when starting up Python to turn it off. Also, you can perform further arbitrary site-specific customizations by adding a sitecustomize module whose import is attempted after the path manipulations have been completed.
- `zipimport` this module allows you to be able to import Python modules that are archived in ZIP files. Note that the functionality in this file is "automagically" called by the importer so there is no need to import this file for use in any application. We mention it here solely as a reference.
- `distutils` this package provides support for building, installing, and distributing Python modules and packages. It also aids in building Python extensions written in C/C++. More information on `distutils` can be found in the Python documentation available at these links:

  http://docs.python.org/dist/dist.html

  http://docs.python.org/inst/inst.html

12.10. Exercises

12-1. *PathSearch versus SearchPath*. What is the difference between a path search and a search path?

12-2. *Importing Attributes*. Assume you have a function called `foo()` in your module `mymodule`.

a.

What are the two ways of importing this function into your namespace for invocation?

b.

What are the namespace implications when choosing one over the other?

12-3. *Importing*. What are the differences between using "**`import`** *`module`*" and "**`from`** *`module`* **`import`** `*`"?

12-4. *Namespaces versus Variable Scope*. How are namespaces and variable scopes different from each other?

12-5. *Using `__import__()`*.

a.

Use `__import__()` to import a module into your namespace. What is the correct syntax you finally used to get it working?

b.

Same as above, but use `__import__()` to import only specific names from modules.

12-6. *Extended Import*. Create a new function called `importAs()`. This function will import a module into your namespace, but with a name you specify, not its original name. For example, calling `newname=importAs ('mymodule')` will import the module `mymodule`, but the module and all its elements are accessible only as `newname` or `newname.attr`. This is the exact functionality provided by the new extended import syntax introduced in Python 2.0.

12-7. *Import Hooks*. Study the import hooks mechanism provided for by the implementation of PEP 302. Implement your own import mechanism, which allows you to obfuscate your Python modules (encryption, bzip2, rot13, etc.) so that the interpreter can decode them properly and import them properly. You may wish to look at how it works with importing zip files (see Section 12.5.7).

PREV NEXT