Python library reference

G. van Rossum

Computer Science/Department of Algorithmics and Architecture

CS-R9524 1995
Python Library Reference

Guido van Rossum

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
guido@cwi.nl

Version 1.2 (10 April 1995)

Abstract

Python is an extensible, interpreted, object-oriented programming language. It supports a wide range of applications, from simple text processing scripts to interactive WWW browsers.

While the Python Reference Manual describes the exact syntax and semantics of the language, it does not describe the standard library that is distributed with the language, and which greatly enhances its immediate usability. This library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs.

This library reference manual documents Python's standard library, as well as many optional library modules (which may or may not be available, depending on whether the underlying platform supports them and on the configuration choices made at compile time). It also documents the standard types of the language and its built-in functions and exceptions, many of which are not or incompletely documented in the Reference Manual.

This manual assumes basic knowledge about the Python language. For an informal introduction to Python, see the Python Tutorial; the Python Reference Manual remains the highest authority on syntactic and semantic questions. Finally, the manual entitled Extending and Embedding the Python Interpreter describes how to add new extensions to Python and how to embed it in other applications.


Keywords & Phrases: Object-oriented languages, Python, libraries, modules, Spanish Inquisition, SPAM.


All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
# Contents

1 Introduction .................................................. 1

2 Built-in Types, Exceptions and Functions ............. 2
   2.1 Built-in Types ........................................... 2
      2.1.1 Truth Value Testing ........................... 2
      2.1.2 Boolean Operations .............................. 3
      2.1.3 Comparisons ....................................... 3
      2.1.4 Numeric Types .................................... 4
      2.1.5 Sequence Types ................................... 6
      2.1.6 Mapping Types ................................. 8
      2.1.7 Other Built-in Types ......................... 8
      2.1.8 Special Attributes ......................... 11
   2.2 Built-in Exceptions ................................... 12
   2.3 Built-in Functions .................................... 14

3 Python Services ............................................... 21
   3.1 Built-in Module sys ................................. 21
   3.2 Standard Module types .............................. 23
   3.3 Standard Module traceback ....................... 25
   3.4 Standard Module pickle .......................... 26
   3.5 Standard Module shelf ............................. 29
   3.6 Standard Module copy .............................. 30
   3.7 Built-in Module marshal ......................... 30
   3.8 Built-in Module imp ............................... 32
      3.8.1 Examples ..................................... 33
   3.9 Built-in Module __builtin__ .................... 34
   3.10 Built-in Module __main__ ...................... 34

4 String Services ................................................ 35
   4.1 Standard Module string ......................... 35
   4.2 Built-in Module regex ........................... 37
   4.3 Standard Module resub .......................... 40
   4.4 Built-in Module struct ....................... 40

5 Miscellaneous Services ........................................ 42
   5.1 Built-in Module math .............................. 42
   5.2 Standard Module rand ............................ 42

6 Generic Operating System Services .............. 45
   6.1 Standard Module os ............................... 45
   6.2 Built-in Module time ............................. 47
   6.3 Standard Module getopt ......................... 48
   6.4 Standard Module tempfile .................... 49

7 Optional Operating System Services .............. 50
   7.1 Built-in Module signal ......................... 50
   7.2 Built-in Module socket ......................... 52
      7.2.1 Socket Objects .............................. 54
      7.2.2 Example ..................................... 56
   7.3 Built-in Module select ......................... 56
   7.4 Built-in Module thread ......................... 57

8 UNIX Specific Services .................................. 59
   8.1 Built-in Module posix ........................... 59
   8.2 Standard Module posixpath .................... 64
   8.3 Built-in Module pwd ............................. 65
   8.4 Built-in Module grp ............................. 66
   8.5 Built-in Module dbm ............................. 66
   8.6 Built-in Module gdbm ......................... 66
   8.7 Built-in Module termios ....................... 67
      8.7.1 Example ..................................... 68
   8.8 Standard Module-TERMIO ....................... 68
   8.9 Built-in Module fcntl ......................... 68
   8.10 Standard Module posixfile .................. 69

9 The Python Debugger .......................................... 72
   9.1 Debugger Commands .............................. 73
   9.2 How It Works ..................................... 75

10 The Python Profiler .......................................... 77
   10.1 Introduction to the profiler ................. 77
   10.2 How Is This Profiler Different From The Old Profiler? 78
   10.3 Instant Users Manual .......................... 78
   10.4 What Is Deterministic Profiling? ........... 80
   10.5 Reference Manual .............................. 80
      10.5.1 The Stats Class ......................... 82
   10.6 Limitations ..................................... 84
   10.7 Calibration ..................................... 84
   10.8 Extensions — Deriving Better Profilers ... 85
      10.8.1 OldProfile Class ....................... 86
      10.8.2 HotProfile Class ....................... 87

11 Internet and WWW Services ...................... 89
11.1 Standard Module `cgi`                     89
11.1.1 Example                                91
11.2 Standard Module `urllib`                 93
11.3 Standard Module `htplib`                 94
11.3.1 HTTP Objects                          95
11.3.2 Example                               95
11.4 Standard Module `ftplib`                 96
11.4.1 FTP Objects                           97
11.5 Standard Module `gopherlib`              99
11.6 Standard Module `nntplib`                99
11.6.1 NNTP Objects                          100
11.7 Standard Module `urlparse`               102
11.8 Standard Module `htmlib`                 103
11.9 Standard Module `smtpplib`               105
11.10 Standard Module `rcp822`                107
11.10.1 Message Objects                      107
11.11 Standard Module `sockets`               108
11.11.1 Additional Methods of Message objects 109

12 Multimedia Services                       110
12.1 Built-in Module `audioop`                110
12.2 Built-in Module `imageio`                113
12.3 Standard Module `aifc`                   114
12.4 Built-in Module `jpeg`                   116
12.5 Built-in Module `rgbimg`                 117

13 Cryptographic Services                   118
13.1 Built-in Module `md5`                    118
13.2 Built-in Module `mpz`                    119
13.3 Built-in Module `rotor`                  120

14 Macintosh Specific Services              122
14.1 Built-in Module `mac`                    122
14.2 Standard Module `macpath`                122
14.3 Built-in Module `etc`                   122
14.3.1 connection object                     123
14.4 Built-in Module `macconsole`            124
14.4.1 macconsole options object             124
14.4.2 console window object                 125
14.5 Built-in Module `macdvr`                 125
14.5.1 dvr result object                     126
14.6 Built-in Module `macfs`                  127
14.6.1 FSSpec objects                        127
14.6.2 alias objects                         128
14.7 Built-in Module `mactcp`                 128
14.7.1 TCP Stream Objects                    129
14.7.2 TCP Status Objects                    130
14.7.3 UDP Stream Objects                    130
14.8 Built-in Module `macspeech`             130
14.8.1 voice objects                         131
14.8.2 speech channel objects                131

15 Standard Windowing Interface              132
15.1 Built-in Module `stdwin`                 132
15.1.1 Functions Defined in Module `stdwin`  132
15.1.2 Window Objects                        135
15.1.3 Drawing Objects                       136
15.1.4 Menu Objects                          138
15.1.5 Bitmap Objects                        138
15.1.6 Text-edit Objects                     139
15.1.7 Example                               140
15.2 Standard Module `stdwinevents`          140
15.3 Standard Module `rect`                  140

16 SGI IRIX Specific Services               142
16.1 Built-in Module `al`                    142
16.1.1 Configuration Objects                143
16.1.2 Port Objects                         143
16.2 Standard Module `am`                    144
16.3 Built-in Module `cd`                    144
16.4 Built-in Module `cl`                    147
16.4.1 Functions Defined in Module `cl`      148
16.4.2 Form Objects                         149
16.4.3 FORMS Objects                        151
16.5 Standard Module `cl`                    152
16.6 Standard Module `fip`                   153
16.7 Built-in Module `fm`                    154
16.8 Built-in Module `gl`                    154
16.9 Standard Modules `GL` and `DEVICE`      156
16.10 Built-in Module `imgfile`              156

17 SunOS Specific Services                  158
17.1 Built-in Module `sunaudiodev`           158
17.1.1 Audio Device Objects                 158
Chapter 1

Introduction

The "Python library" contains several different kinds of components.

It contains data types that would normally be considered part of the "core" of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an import statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built into the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, like socket I/O; others provide interfaces that are specific to a particular application domain, like the World-Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized "from the inside out": it first describes the built-in data types, then the built-in functions and exceptions, and finally the modules, grouped in chapters of related modules. The ordering of the chapters as well as the ordering of the modules within each chapter is roughly from most relevant to least important.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don't have to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module random) and read a section or two.

Let the show begin!

Chapter 2

Built-in Types, Exceptions and Functions

Names for built-in exceptions and functions are found in a separate symbol table. This table is searched last when the interpreter looks up the meaning of a name, so local and global user-defined names can override built-in names. Built-in types are described together here for easy reference.¹

The tables in this chapter document the priorities of operators by listing them in order of ascending priority (within a table) and grouping operators that have the same priority in the same box. Binary operators of the same priority group from left to right. (Unary operators group from right to left, but there you have no real choice.) See Chapter 5 of the Python Reference Manual for the complete picture on operator priorities.

2.1 Built-in Types

The following sections describe the standard types that are built into the interpreter. These are the numeric types, sequence types, and several others, including types themselves. There is no explicit Boolean type; use integers instead.

Some operations are supported by several object types; in particular, all objects can be compared, tested for truth value, and converted to a string (with the `str` notation). The latter conversion is implicitly used when an object is written by the print statement.

2.1.1 Truth Value Testing

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below. The following values are considered false:

- None
- zero of any numeric type, e.g., 0, 0L, 0.0
- any empty sequence, e.g., '', (), []

¹Most descriptions sorely lack explanations of the exceptions that may be raised — this will be fixed in a future version of this manual.
• any empty mapping, e.g., {}.
• instances of user-defined classes, if the class defines a __nonzero__() or __len__() method, when that method returns zero.

All other values are considered true — so objects of many types are always true.

Operations and built-in functions that have a Boolean result always return 0 for false and 1 for true, unless otherwise stated. (Important exception: the Boolean operations 'or' and 'and' always return one of their operands.)

2.1.2 Boolean Operations

These are the Boolean operations, ordered by ascending priority:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x or y</code></td>
<td>if <code>x</code> is false, then <code>x</code>, else <code>y</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>x and y</code></td>
<td>if <code>x</code> is false, then <code>x</code>, else <code>y</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>not x</code></td>
<td>if <code>x</code> is false, then 1, else 0</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:
(1) These only evaluate their second argument if needed for their outcome.
(2) 'not' has a lower priority than non-Boolean operators, so e.g. `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

2.1.3 Comparisons

Comparison operations are supported by all objects. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily, e.g. `x < y <= z` is equivalent to `x < y` and `y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>strictly less than</td>
<td></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal</td>
<td></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>strictly greater than</td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal</td>
<td></td>
</tr>
<tr>
<td><code>==</code></td>
<td>equal</td>
<td></td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal</td>
<td>(1)</td>
</tr>
<tr>
<td><code>is</code></td>
<td>object identity</td>
<td></td>
</tr>
<tr>
<td><code>is not</code></td>
<td>negated object identity</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
(1) `<>` and `!=` are alternate spellings for the same operator. (I couldn’t choose between ABC and C! :

Objects of different types, except different numeric types, never compare equal; such objects are ordered consistently but arbitrarily (so that sorting a heterogeneous array yields a consistent result). Furthermore, some types (e.g., windows) support only a degenerate notion of comparison where any two objects of that type are unequal. Again, such objects are ordered arbitrarily but consistently.

(Implementation note: objects of different types except numbers are ordered by their type names; objects of the same types that don’t support proper comparison are ordered by their address.)

Two more operations with the same syntactic priority, `in` and `not in`, are supported only by sequence types (below).

2.1.4 Numeric Types

There are three numeric types: plain integers, long integers, and floating point numbers. Plain integers (also just called integers) are implemented using long in C, which gives them at least 32 bits of precision. Long integers have unlimited precision. Floating point numbers are implemented using double in C. All bets on their precision are off unless you happen to know the machine you are working with.

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex and octal numbers) yield plain integers. Integer literals with an `l` or `L` suffix yield long integers (`L` is preferred because `1l` looks too much like eleven!). Numeric literals containing a decimal point or an exponent sign yield floating point numbers.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "smaller" type is converted to that of the other, where plain integer is smaller than long integer is smaller than floating point. Comparisons between numbers of mixed type use the same rule.\(^{3}\) The functions `int()`, `long()` and `float()` can be used to coerce numbers to a specific type.

All numeric types support the following operations, sorted by ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

\(^{3}\)As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similar for tuples.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x + y</td>
<td>sum of x and y</td>
<td></td>
</tr>
<tr>
<td>x - y</td>
<td>difference of x and y</td>
<td></td>
</tr>
<tr>
<td>x * y</td>
<td>product of x and y</td>
<td></td>
</tr>
<tr>
<td>x / y</td>
<td>quotient of x and y</td>
<td></td>
</tr>
<tr>
<td>x % y</td>
<td>remainder of x / y</td>
<td>(1)</td>
</tr>
<tr>
<td>-x</td>
<td>x negated</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x unchanged</td>
<td></td>
</tr>
<tr>
<td>abs(s)</td>
<td>absolute value of s</td>
<td></td>
</tr>
<tr>
<td>int(s)</td>
<td>x converted to integer</td>
<td>(2)</td>
</tr>
<tr>
<td>long(s)</td>
<td>x converted to long integer</td>
<td>(2)</td>
</tr>
<tr>
<td>float(x)</td>
<td>x converted to floating point</td>
<td></td>
</tr>
<tr>
<td>divmod(x, y)</td>
<td>the pair (x / y, x % y)</td>
<td></td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>x to the power y</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Notes:

(1) For (plain or long) integer division, the result is an integer; it always truncates towards zero.

(2) Conversion from floating point to (long or plain) integer may round or truncate as in C; see functions floor() and ceil() in module math for well-defined conversions.

(3) See the section on built-in functions for an exact definition.

### Bit-string Operations on Integer Types

Plain and long integer types support additional operations that make sense only for bit-strings. Negative numbers are treated as their 2’s complement value (for long integers, this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bit-wise operations are all lower than the numeric operations and higher than the comparisons; the unary operation ‘ ‘ has the same priority as the other unary numeric operations (‘+’ and ‘-’).

This table lists the bit-string operations sorted in ascending priority (operations in the same box have the same priority):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>bitwise or of x and y</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>bitwise exclusive or of x and y</td>
<td></td>
</tr>
<tr>
<td>x &amp; y</td>
<td>bitwise and of x and y</td>
<td></td>
</tr>
<tr>
<td>x &lt;&lt; n</td>
<td>x shifted left by n bits</td>
<td>(1), (2)</td>
</tr>
<tr>
<td>x &gt;&gt; n</td>
<td>x shifted right by n bits</td>
<td>(1), (3)</td>
</tr>
<tr>
<td>x</td>
<td>the bits of x inverted</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) Negative shift counts are illegal.

(2) A left shift by n bits is equivalent to multiplication by pow(2, n) without overflow check.

(3) A right shift by n bits is equivalent to division by pow(2, n) without overflow check.

### 2.1.5 Sequence Types

There are three sequence types: strings, lists and tuples.

Strings literals are written in single or double quotes: ‘xyzzy’, "foroboss". See Chapter 2 of the Python Reference Manual for more about string literals. Lists are constructed with square brackets, separating items with commas: [a, b, c]. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, e.g., (a,).

Sequence types support the following operations. The ‘in’ and ‘not in’ operations have the same priorities as the comparison operations. The ‘+’ and ‘*’ operations have the same priority as the corresponding numeric operations.3

This table lists the sequence operations sorted in ascending priority (operations in the same box have the same priority). In the table, s and t are sequences of the same type; n, i and j are integers:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>s in s</td>
<td>1 if an item of s is equal to x, else 0</td>
<td></td>
</tr>
<tr>
<td>x not in s</td>
<td>0 if an item of s is equal to x, else 1</td>
<td></td>
</tr>
<tr>
<td>s + t</td>
<td>the concatenation of s and t</td>
<td></td>
</tr>
<tr>
<td>s * n</td>
<td>n copies of s concatenated</td>
<td></td>
</tr>
<tr>
<td>s(t[i])</td>
<td>tth item of s, origin 0</td>
<td>(1)</td>
</tr>
<tr>
<td>s(i:j)</td>
<td>slice of s from i to j</td>
<td>(1), (2)</td>
</tr>
<tr>
<td>len(s)</td>
<td>length of s</td>
<td></td>
</tr>
<tr>
<td>min(s)</td>
<td>smallest item of s</td>
<td></td>
</tr>
<tr>
<td>max(s)</td>
<td>largest item of s</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) If i or j is negative, the index is relative to the end of the string, i.e., len(s) + i or len(s) + j is substituted. But note that -0 is still 0.

(2) The slice of s from i to j is defined as the sequence of items with index k such that i <= k < j.

More String Operations

String objects have one unique built-in operation: the % operator (modulo) with a string left argument interprets this string as a C printf format string to be applied to the right argument, and returns the string resulting from this formatting operation.

3They must have since the parser can’t tell the type of the operands.
The right argument should be a tuple with one item for each argument required by the format string; if the string requires a single argument, the right argument may also be a single non-tuple object. The following format characters are understood: %, c, s, i, d, u, o, x, X, e, E, f, g, G. Width and precision may be a * to specify that an integer argument specifies the actual width or precision. The flag characters -, +, blank, # and 0 are understood. The size specifiers h, l or L do not assume that an integer argument specifies the actual width or precision. The ANSI features %p and %n are not supported. Since Python strings have an explicit length, %s conversions don’t assume that ‘\0’ is the end of the string.

For safety reasons, floating point precisions are clipped to 50; %f conversions for numbers whose absolute value is over 1e25 are replaced by %g conversions. All other errors raise exceptions.

If the right argument is a dictionary (or any kind of mapping), then the formats in the string must have a parenthesized key into that dictionary inserted immediately after the % character, and each format in the string corresponds to the corresponding entry from the mapping. E.g.

```python
>>> count = 2
>>> language = 'Python'
>>> print '{0} languages have {1} quote types.' % vars()
Python has 002 quote types.
```

In this case no * specifiers may occur in a format (since they require sequential parameter list).

Additional string operations are defined in standard module string and in built-in module regex.

### Mutable Sequence Types

List objects support additional operations that allow in-place modification of the object. These operations would be supported by other mutable sequence types (when added to the language) as well. Strings and tuples are immutable sequence types and such objects cannot be modified once created.

The following operations are defined on mutable sequence types (where `x` is an arbitrary object):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[i] = x</code></td>
<td><code>s[i]</code> is replaced by <code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>s[i:j] = t</code></td>
<td><code>s[i:j]</code> is replaced by <code>t</code></td>
<td></td>
</tr>
<tr>
<td><code>del s[i]</code></td>
<td>same as <code>s[i] = []</code></td>
<td></td>
</tr>
<tr>
<td><code>s.append(x)</code></td>
<td>same as <code>s[len(s):len(s)] = [x]</code></td>
<td></td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>return number of <code>x</code> for which <code>s[i] == x</code></td>
<td></td>
</tr>
<tr>
<td><code>s.index(x)</code></td>
<td>return smallest <code>i</code> such that <code>s[i] == x</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>s.insert(i, x)</code></td>
<td>same as <code>s[i] = x</code></td>
<td></td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td>same as <code>del s[i]</code></td>
<td></td>
</tr>
<tr>
<td><code>s.reverse()</code></td>
<td>reverses the items of <code>s</code> in place</td>
<td></td>
</tr>
<tr>
<td><code>s.sort()</code></td>
<td>permutes the items of <code>s</code> to satisfy <code>s[i] &lt;= s[j]</code> for <code>i &lt; j</code></td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

1. Raises an exception if `i` is not in `s`.
2. Keys and values are listed in random order.

### 2.1.6 Mapping Types

A mapping object maps values of one type (the key type) to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the `dict` object. A dictionary’s keys are almost always strings. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g. 1 and 1.0) then they can be used interchangeably to index the same dictionary entry.

Dictionaries are created by placing a comma-separated list of key: value pairs within braces, for example: `{'jack': 4096, 'sjoerd': 4127}` or `{4096: 'jack', 4127: 'sjoerd'}`.

The following operations are defined on mappings (where `a` is a mapping, `k` is a key and `x` is an arbitrary object):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>the number of items in <code>a</code></td>
<td></td>
</tr>
<tr>
<td><code>a[k]</code></td>
<td>the item of <code>a</code> with key <code>k</code></td>
<td></td>
</tr>
<tr>
<td><code>a[k] = x</code></td>
<td>set <code>a[k]</code> to <code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>del a[k]</code></td>
<td>remove <code>a[k]</code> from <code>a</code></td>
<td></td>
</tr>
<tr>
<td><code>a.items()</code></td>
<td>a copy of <code>a</code>’s list of (key, item) pairs</td>
<td></td>
</tr>
<tr>
<td><code>a.keys()</code></td>
<td>a copy of <code>a</code>’s list of keys</td>
<td>(1)</td>
</tr>
<tr>
<td><code>a.values()</code></td>
<td>a copy of <code>a</code>’s list of values</td>
<td></td>
</tr>
<tr>
<td><code>a.has_key(k)</code></td>
<td>1 if <code>a</code> has a key <code>k</code>, else 0</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

1. Raises an exception if `k` is not in the map.
2. Keys and values are listed in random order.

### 2.1.7 Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.
Modules

The only special operation on a module is attribute access: \texttt{m.name}, where \texttt{m} is a module and \texttt{name} accesses a name defined in \texttt{m}'s symbol table. Module attributes can be assigned to. (Note that the \texttt{import} statement is not, strictly spoken, an operation on a module object; \texttt{import foo} does not require a module object named \texttt{foo} to exist, rather it requires an (external) \texttt{definition} for a module named \texttt{foo} somewhere.)

A special member of every module is \texttt{__dict__}. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the \texttt{__dict__} attribute is not possible (i.e. you can write \texttt{m.__dict__['a'] = 1}, which defines \texttt{m.a} to be \texttt{1}, but you can't write \texttt{m.__dict__ = {}}. Modules are written like this: \texttt{<module 'sys'>}.

Classes and Class Instances

(See Chapters 3 and 7 of the Python Reference Manual for these.)

Functions

Function objects are created by function definitions. The only operation on a function object is to call it: \texttt{func(args, ...)}. There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

The implementation adds two special read-only attributes: \texttt{f.func_code} is a function's code object (see below) and \texttt{f.func_globals} is the dictionary used as the function's global name space (this is the same as \texttt{m.__dict__} where \texttt{m} is the module in which the function \texttt{f} was defined).

Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as \texttt{append()} on lists) and class instance methods. Built-in methods are described with the types that support them.

The implementation adds two special read-only attributes to class instance methods: \texttt{m.im_self} is the object whose method this is, and \texttt{m.im_func} is the function implementing the method. Calling \texttt{m(arg-1, arg-2, ..., arg-n)} is completely equivalent to calling \texttt{m.im_func(m.im_self, arg-1, arg-2, ..., arg-n)}.

(See the Python Reference Manual for more info.)

Code Objects

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in \texttt{compile()} function and can be extracted from function objects through their \texttt{func_code} attribute.

A code object can be executed or evaluated by passing it (instead of a source string) to the \texttt{exec} statement or the built-in \texttt{eval()} function.

(See the Python Reference Manual for more info.)

Type Objects

Type objects represent the various object types. An object's type is accessed by the built-in \texttt{type()} function. There are no special operations on types. The standard module \texttt{types} defines names for all standard built-in types.

Types are written like this: \texttt{<type 'int'>}.

The Null Object

This object is returned by functions that don't explicitly return a value. It supports no special operations. There is exactly one null object, named \texttt{None} (a built-in name).

It is written as \texttt{None}.

File Objects

File objects are implemented using C's \texttt{stdio} package and can be created with the built-in function \texttt{open()} described under Built-in Functions below. They are also returned by some other built-in functions and methods, e.g. \texttt{posix.popen() and posix.fdopen()} and the \texttt{makefile()} method of socket objects.

When a file operation fails for an I/O-related reason, the exception \texttt{IOError} is raised. This includes situations where the operation is not defined for some reason, like \texttt{seek()} on a tty device or writing a file opened for reading.

Files have the following methods:

\begin{itemize}
  \item \texttt{close()} - Close the file. A closed file cannot be read or written anymore.
  \item \texttt{flush()} - Flush the internal buffer, like \texttt{stdio's flush()}.
  \item \texttt{isatty()} - Return 1 if the file is connected to a tty-like device, else 0.
  \item \texttt{read(size)} - Read at most \texttt{size} bytes from the file (less if the read hits EOF or no more data is immediately
available on a pipe, tty or similar device). If the *size* argument is omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.)

**readline()**

Read one entire line from the file. A trailing newline character is kept in the string\(^6\) (but may be absent when a file ends with an incomplete line). An empty string is returned when EOF is hit immediately. Note: unlike stdio's fgets(), the returned string contains null characters ("\0") if they occurred in the input.

**readlines()**

Read until EOF using readline() and return a list containing the lines thus read.

### 2.2 Built-in Exceptions

Exceptions are string objects. Two distinct string objects with the same value are different exceptions. This is done to force programmers to use exception names rather than their string value when specifying exception handlers. The string value of all built-in exceptions is their name, but this is not a requirement for user-defined exceptions or exceptions defined by library modules.

The following exceptions can be generated by the interpreter or built-in functions. Except where mentioned, they have an 'associated value' indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code).

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition 'just like' the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

#### AttributeError

Raised when an attribute reference or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)

#### EOFError

Raised when one of the built-in functions (input() or raw_input()) hits an end-of-file condition (EOF) without reading any data. (N.B.: the read() and readline() methods of file objects return an empty string when they hit EOF.) No associated value.

#### IOError

Raised when an I/O operation (such as a print statement, the built-in open() function or a method of a file object) fails for an I/O-related reason, e.g., 'file not found', 'disk full'.

#### ImportError

Raised when an import statement fails to find the module definition or when a from ... import fails to find a name that is to be imported.

#### IndexError

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, TypeError is raised.)

#### KeyError

Raised when a mapping (dictionary) key is not found in the set of existing keys.

#### KeyboardInterrupt

Raised when the user hits the interrupt key (normally Control-C or DEL). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function input() or raw_input() is waiting for input also raise this exception. No associated value.

#### MemoryError

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's...
malloc() function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

**NameError**
Raised when a local or global name is not found. This applies only to unqualified names. The associated value is the name that could not be found.

**OverflowError**
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise MemoryError than give up). Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren’t checked. For plain integers, all operations that can overflow are checked except left shift, where typical applications prefer to drop bits than raise an exception.

**RuntimeError**
Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong. (This exception is a relic from a previous version of the interpreter; it is not used any more except by some extension modules that haven’t been converted to define their own exceptions yet.)

**SyntaxError**
Raised when the parser encounters a syntax error. This may occur in an import statement, in an exec statement, in a call to the built-in function eval() or input(), or when reading the initial script or standard input (also interactively).

**SystemError**
Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version string of the Python interpreter (sys.version; it is also printed at the start of an interactive Python session), the exact error message (the exception’s associated value) and if possible the source of the program that triggered the error.

**SystemExit**
This exception is raised by the sys.exit() function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C’s exit() function); if it is None, the exit status is zero; if it has another type (such as a string), the object’s value is printed and the exit status is one.

A call to sys.exit is translated into an exception so that clean-up handlers (finally clauses of try statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The posix._exit() function can be used if it is absolutely positively necessary to exit immediately (e.g., after a fork() in the child process).

**TypeError**
Raised when a built-in operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

**ValueError**
Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.

**ZeroDivisionError**
Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

### 2.3 Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

- **abs()**
  Return the absolute value of a number. The argument may be a plain or long integer or a floating point number.

- **apply(function, args)**
  The function argument must be a callable object (a user-defined or built-in function or method, or a class object) and the args argument must be a tuple. The function is called with args as argument list; the number of arguments is the the length of the tuple. (This is different from just calling func(args), since in that case there is always exactly one argument.)

- **chr(i)**
  Return a string of one character whose ASCII code is the integer i.e., chr(97) returns the string ‘a’. This is the inverse of ord(). The argument must be in the range 0..255, inclusive.

- **cmp(x, y)**
  Compare the two objects x and y and return an integer according to the outcome. The return value is negative if 
  
  - x < y, zero if x == y and strictly positive if x > y.

- **coerce(x, y)**
  Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations.

- **compile(string, filename, kind)**
  Compile the string into a code object. Code objects can be executed by an exec statement or evaluated by a call to eval(). The filename argument should give the file from which the code was read; pass e.g. <'<string>' if it wasn’t read from a file. The kind argument specifies what kind of code must be compiled; it can be ‘exec’ if string consists of a sequence of statements, or ‘eval’ if it consists of a single expression.

- **delattr(object, name)**
  This is a relative of setattr. The arguments are an object and a string. The string must be the name of one of the object’s attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, ‘foo.bar’) is equivalent to del x.foo.bar.

- **dir()**
  Without arguments, return the list of names in the current local symbol table. With a module, class or class instance object as argument (or anything else that has a __dict__ attribute), returns the list of names in that object’s attribute dictionary. The resulting list is sorted. For
example:

```python
>>> import sys
>>> dir()
['sys']
>>> dir(sys)
['argv', 'exit', 'modules', 'path', 'stderr', 'stdin', 'stdout']
```

divmod(a, b)

Take two numbers as arguments and return a pair of integers consisting of their integer quotient and remainder. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as \((a / b, a \% b)\). For floating point numbers the result is the same as \((\text{math.floor}(a / b), a \% b)\).

eval(expression, globals, locals)

The arguments are a string and two optional dictionaries. The \textit{expression} argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the \textit{globals} and \textit{locals} dictionaries as global and local name space. If the \textit{locals} dictionary is omitted it defaults to the \textit{globals} dictionary. If both dictionaries are omitted, the expression is executed in the environment where \textit{eval} is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```python
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (e.g. created by \textit{compile}()). In this case pass a code object instead of a string. The code object must have been compiled passing \textit{eval} to the \textit{kind} argument.

Hints: dynamic execution of statements is supported by the \textit{exec} statement. Execution of statements from a file is supported by the \textit{execfile()} function. The \textit{vars()} function returns the current local dictionary, which may be useful to pass around for use by \textit{eval()} or \textit{execfile()}.

execfile(file, globals, locals)

This function is similar to the \textit{exec} statement, but parses a file instead of a string. It is different from the \textit{import} statement in that it does not use the module administration — it reads the file unconditionally and does not create a new module.\(^7\) The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the \textit{globals} and \textit{locals} dictionaries as global and local name space. If the \textit{locals} dictionary is omitted it defaults to the \textit{globals} dictionary. If both dictionaries are omitted, the expression is executed in the environment where \textit{execfile()} is called. The return value is \textit{None}.

\(^7\)This is used relatively rarely so does not warrant being made into a statement.

filter(function, list)

Construct a list from those elements of \textit{list} for which \textit{function} returns true. If \textit{list} is a string or a tuple, the result also has that type; otherwise it is always a list. If \textit{function} is \textit{None}, the identity function is assumed, i.e. all elements of \textit{list} that are false (zero or empty) are removed.

float(x)

Convert a number to floating point. The argument may be a plain or long integer or a floating point number.

getattr(object, name)

The arguments are an object and a string. The string must be the name of one of the object's attributes. The result is the value of that attribute. For example, \texttt{getattr(x, 'foobar')} is equivalent to \texttt{x.foobar}.

hash(object)

The arguments are an object and a string. The result is 1 if the string is the name of one of the object's attributes, 0 if not. (This is implemented by calling \texttt{getattr(object, name)} and seeing whether it raises an exception or not).

hex(x)

Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression.

id(object)

Return the 'identity' of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. (Two objects whose lifetimes are disjoint may have the same id() value.) (Implementation note: this is the address of the object.)

input(prompt)

Almost equivalent to \textit{eval(raw_input(prompt))}. Like \textit{raw_input()}, the \textit{prompt} argument is optional. The difference is that a long input expression may be broken over multiple lines using the backslash convention.

int(x)

Convert a number to a plain integer. The argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers is defined by the C semantics; normally the conversion truncates towards zero.\(^8\)

len(s)

Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

long(x)

Convert a number to a long integer. The argument may be a plain or long integer or a floating point number.\(^8\)

\(^8\)This is ugly — the language definition should require truncation towards zero.
map(function, list,...)
Apply function to every item of list and return a list of the results. If additional list arguments are passed, function must take that many arguments and is applied to the items of all lists in parallel; if a list is shorter than another it is assumed to be extended with None items. If function is None, the identity function is assumed; if there are multiple list arguments, map returns a list consisting of tuples containing the corresponding items from all lists (i.e. a kind of transpose operation). The list arguments may be any kind of sequence; the result is always a list.

max(s)
Return the largest item of a non-empty sequence (string, tuple or list).

min(s)
Return the smallest item of a non-empty sequence (string, tuple or list).

ord(x)
Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

open(filename [, mode [, bufsize]])
Return a new file object (described earlier under Built-in Types). The first two arguments are the same as for stdio's fopen(): filename is the file name to be opened, mode indicates how the file is to be opened: 'r' for reading, 'w' for writing (truncating an existing file), and 'a' opens it for appending. Modes 'r+', 'w+' and 'a+' open the file for updating, provided the underlying stdio library understands this. On systems that differentiate between binary and text files, 'b' appended to the mode opens the file in binary mode. If the file cannot be opened, EOFError is raised. If mode is omitted, it defaults to 'r'. The optional bufsize argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative bufsize means to use the system default, which is usually line buffered for tty devices and fully buffered for other files.9

pow(x, y [, z])
Return x to the power y; if z is present, return x to the power y, modulo z (computed more efficiently than pow(x, y, z)). The arguments must have numeric types. With mixed operand types, the rules for binary arithmetic operators apply. The effective operand type is also the type of the result; if the result is not expressible in this type, the function raises an exception; e.g., pow(2, -1) or pow(2, 35000) is not allowed.

range([start], [end], [step])
This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start+step, start+2*step, ...]. If step is positive, the last element is the largest start + i * step less than end; if step is negative, the last element is the largest start + i * step greater than end. step must not be zero (or else an exception is raised). Example:

```python
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[0]
>>> range(1, 0)
[]
```

raw_input(prompt)
If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. Example:

```python
>>> s = raw_input(' --> ')
---> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```
When a module is reloaded, its dictionary (containing the module’s global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module’s advantage if it maintains a global table or cache of objects — with a try statement it can test for the table’s presence and skip its initialization if desired.

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for `sys`, `__main__`, and `__builtin__`. In certain cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (module name) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

```python
>>> import types

Rede®nitions of names will o v erride the old de®nitions, so this is generally not a problem. If the ne w v ersion of a module does not de®ne a name that was de®ned by the old v ersion, the old de®nition remains. This feature can be used to the module’s adv antage if it maintains a global
table or cache of objects Ð with a try statement it can test for the table’s presence and skip its initialization if desired.

It is le gal though generally not v ery useful to reload b uilt-in or dynamically loaded modules, except for `sys`, `__main__`, and `__builtin__`. In certain cases, ho wever, e xtension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ... import ...`, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (module name) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

```python
repl (object)

Return a string containing a printable representation of an object. This is the same value yielded
by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an
ordinary function. For many types, this function makes an attempt to return a string that would
yield an object with the same value when passed to `eval()`.

round (x, n)

Return the floating point value x rounded to n digits after the decimal point. If n is omitted,
it defaults to zero. The result is a floating point number. Values are rounded to the closest
multiple of 10 to the power minus n; if two multiples are equally close, rounding is done away
from 0 (so e.g. `round(0.5)` is 1.0 and `round(-0.5)` is -1.0).

setattr (object, name, value)

This is the counterpart of `getattr`. The arguments are an object, a string and an arbitrary value.
The string must be the name of one of the object’s attributes. The function assigns the value to
the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is
equivalent to `x.foobar = 123`.

str (object)

Return a string containing a nicely printable representation of an object. For strings, this returns
the string itself. The difference with `repr (object)` is that `str (object)` does not always
attempt to return a string that is acceptable to `eval()`; its goal is to return a printable string.

tuple (sequence)

Return a tuple whose items are the same and in the same order as `sequence`’s items. If `sequence`
is already a tuple, it is returned unchanged. For instance, `tuple(‘abc’)` returns returns
(‘a’, ‘b’, ‘c’) and `tuple([1, 2, 3])` returns (1, 2, 3).

type (object)

Return the type of an object. The return value is a type object. The standard module `types`
defines names for all built-in types. For instance:

```
>>> import types
>>> if type(x) == types.StringType: print ”It’s a string”
```

vars (object)

Without arguments, return a dictionary corresponding to the current local symbol table. With a
module, class or instance object as argument (or anything else that has a `__dict__` attribute), returns a dictionary corresponding to the object’s symbol table. The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined.

xrange (start, end [, step ])

This function is very similar to `range()`, but returns an “xrange object” instead of a list. This
is an opaque sequence type which yields the same values as the corresponding list, without
actually storing them all simultaneously. The advantage of `xrange()` over `range()` is
minimal (since `xrange()` still has to create the values when asked for them) except when
a very large range is used on a memory-starved machine (e.g. MS-DOS) or when all of the
range’s elements are never used (e.g. when the loop is usually terminated with `break`).

---

10In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from
other scopes (e.g. modules) can be. This may change.
Chapter 3

Python Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here’s an overview:

**sys**  
Access system specific parameters and functions.

**types**  
Names for all built-in types.

**traceback**  
Print or retrieve a stack traceback.

**pickle**  
Convert Python objects to streams of bytes and back.

**shelve**  
Python object persistence.

**copy**  
Shallow and deep copy operations.

**marshal**  
Convert Python objects to streams of bytes and back (with different constraints).

**imp**  
Access the implementation of the import statement.

**__builtin__**  
The set of built-in functions.

**__main__**  
The environment where the top-level script is run.

### 3.1 Built-in Module sys

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available.

**argv**  
The list of command line arguments passed to a Python script. `sys.argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `sys.argv[0]` is set to the string "-c". If no script name was passed to the Python interpreter, `sys.argv` has zero length.

**exc_type**  
**exc_value**  
**exc_traceback**  
These three variables are not always defined; they are set when an exception handler (an except clause of a try statement) is invoked. Their meaning is: `exc_type` gets the exception type of the exception being handled; `exc_value` gets the exception parameter (its associated value or the second argument to `raise`); `exc_traceback` gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

**exit(n)**  
Exit from Python with numeric exit status `n`. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by `finally` clauses of `try` statements are honored, and it is possible to catch the exit attempt at an outer level.

**exitfunc**  
This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits in any way (except not when a fatal error occurs: in that case the interpreter’s internal state cannot be trusted).

**last_type**  
**last_value**  
**last_traceback**  
These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error (which may be hard to reproduce). The meaning of the variables is the same as that of `exc_type`, `exc_value` and `exc_traceback`, respectively.

**modules**  
Gives the list of modules that have already been loaded. This can be manipulated to force re-loading of modules and other tricks.

**path**  
A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, or an installation-dependent default.

**ps1**  
**ps2**  
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are ‘>>> ’ and ‘... ’.
setcheckinterval(interval)

Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 10, meaning the check is performed every 10 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value ≤ 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

settrace(tracefunc)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. See section "How It Works" in the chapter on the Python Debugger.

setprofile(profilefunc)

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See the chapter on the Python Profiler. The system's profile function is called similarly to the system's trace function (see sys.settrace), but it isn't called for each executed line of code (only on call and return and when an exception occurs). Also, its return value is not used, so it can just return None.

stdin

stdout

stderr

File objects corresponding to the interpreter's standard input, output and error streams.

sys.stdin is used for all interpreter input except for scripts but including calls to input() and raw_input(). sys.stdout is used for the output of print and expression statements and for the prompts of input() and raw_input(). The interpreter's own prompts and (almost all of) its error messages go to sys.stderr. sys.stdin and sys.stdout needn't be built-in file objects: any object is acceptable as long as it has a write method that takes a string argument. (Changing these objects doesn't affect the standard I/O streams of processes executed by popen(), system() or the exec*() family of functions in the os module.)

tracebacklimit

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is 1000. When set to 0 or less, all traceback information is suppressed and only the exception type and value are printed.

3.2 Standard Module types

This module defines names for all object types that are used by the standard Python interpreter (but not for the types defined by various extension modules). It is safe to use "from types import *" — the module does not export any other names besides the ones listed here. New names exported by future versions of this module will all end in Type.

Typical use is for functions that do different things depending on their argument types, like the following:

```python
from types import *

def delete(list, item):
    if type(item) is IntType:
        del list[item]
    else:
        list.remove(item)
```

The module defines the following names:

NoneType

The type of None.

TypeError

The type of type objects (such as returned by type()).

IntType

The type of integers (e.g. 1).

LongType

The type of long integers (e.g. 1L).

FloatType

The type of floating point numbers (e.g. 1.0).

 StringType

The type of character strings (e.g. 'Spam').

TupleType

The type of tuples (e.g. (1, 2, 3, 'Spam')).

ListType

The type of lists (e.g. [0, 1, 2, 3]).

DictType

The type of dictionaries (e.g. {'Bacon': 1, 'Ham': 0}).

DictionaryType

An alternative name for DictType.

FunctionType

The type of user-defined functions and lambdas.

LambdaType

An alternative name for FunctionType.

CodeType

The type for code objects such as returned by compile().

classType

The type of user-defined classes.

InstanceType

The type of instances of user-defined classes.

MethodType

The type of methods of user-defined class instances.

UnboundMethodType
An alternative name for `MethodType`.

`BuiltInFunctionType`
- The type of built-in functions like `len` or `sys.exit`.

`BuiltInMethodType`
- An alternative name for `BuiltInFunction`.

`ModuleType`
- The type of modules.

`FileType`
- The type of open file objects such as `sys.stdout`.

`XRangeType`
- The type of range objects returned by `range()`.

`TracebackType`
- The type of traceback objects such as found in `sys.exc_traceback`.

`FrameType`
- The type of frame objects such as found in `tb.tb_frame` if `tb` is a traceback object.

### 3.3 Standard Module `traceback`

This module provides a standard interface to format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, e.g. in a “wrapper” around the interpreter.

The module uses traceback objects — this is the object type that is stored in the variables `sys.exc_traceback` and `sys.last_traceback`.

The module defines the following functions:

- `print_tb(traceback[, limit])`
  - Print up to `limit` stack trace entries from `traceback`. If `limit` is omitted or `None`, all entries are printed.

- `extract_tb(traceback[, limit])`
  - Return a list of up to `limit` “pre-processed” stack trace entries extracted from `traceback`. It is useful for alternate formatting of stack traces. If `limit` is omitted or `None`, all entries are extracted. A “pre-processed” stack trace entry is a quadruple (`filename`, `line number`, `function name`, `line text`) representing the information that is usually printed for a stack trace.

- `print_exception(type, value, traceback[, limit])`
  - Print exception information and up to `limit` stack trace entries from `traceback`. This differs from `print_tb` in the following ways: (1) if `traceback` is not `None`, it prints a header “`Traceback (most recent):`”;
  - (2) it prints the exception `type` and `value` after the stack trace;
  - (3) it prints `SyntaxError` if `value` and `SyntaxError` has the appropriate format, it prints the line where the syntax error occurred with a caret indication the approximate position of the error.

- `print_exc([limit])`
  - This is a shorthand for `print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit)`.

- `print_last([limit])`
  - This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit)`.

### 3.4 Standard Module `pickle`

The `pickle` module implements a basic but powerful algorithm for “pickling” (a.k.a. serializing, marshalling or flattening) nearly arbitrary Python objects. This is the act of converting objects to a stream of bytes (and back: “unpickling”). This is a more primitive notion than persistence — although `pickle` reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) area of concurrent access to persistent objects. The `pickle` module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. The most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The `pickle` module also provides a simple interface to pickle and unpickle objects on “dbm”-style database files.

Unlike the built-in module `marshal`, `pickle` handles the following correctly:

- recursive objects (objects containing references to themselves)
- object sharing (references to the same object in different places)
- user-defined classes and their instances

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as CORBA (which probably can’t represent pointer sharing or recursive objects); however, it means that non-Python programs may not be able to reconstruct pickled Python objects.

The `pickle` data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. However, small integers actually take less space when represented as minimal-size decimal strings than when represented as 32-bit binary numbers, and strings are only much longer if the they contain many control characters or 8-bit characters. The big advantage of using printable ASCII (and of some other characteristics of `pickle`’s representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor. (I could have gone a step further and used a notation like S-expressions, but the parser (currently written in Python) would have been considerably more complicated and slower, and the files would probably have become much larger.)

The `pickle` module doesn’t handle code objects, which the `marshal` module does. I suppose `pickle` could, and maybe it should, but there’s probably no great need for it right now (as long as `marshal` continues to be used for reading and writing code objects), and at least this avoids the possibility of smuggling Trojan horses into a program.
For the benefit of persistence modules written using pickle, it supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a name, which is an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the pickle module — the persistent object module will have to implement a method `persistent_load()` to write references to persistent objects, the persistent module must define a method `persistent_id()` which returns either `None` or the persistent ID of the object.

There are some restrictions on the pickling of class instances. First of all, the class must be defined at the top level in a module.

Next, it must normally be possible to create class instances by calling the class without arguments. If this is undesirable, the class can define a method `__getinitargs__()` which should return a `tuple` containing the arguments to be passed to the class constructor `__init__()`.

Classes can further influence how their instances are pickled — if the class defines the method `__getstate__()` it is called and the return state is pickled as the contents for the instance, and if the class defines the method `__setstate__()` it is called with the unpickled state. (Note that these methods can also be used to implement copying class instances.) If there is no `__getstate__()` method, the instance's `__dict__` is pickled. If there is no `__setstate__()` method, the pickled object must be a dictionary and its items are assigned to the new instance's dictionary. (If a class defines both `__getstate__()` and `__setstate__()`, the state object needn't be a dictionary — these methods can do what they want.) This protocol is also used by the shallow and deep copying operations defined in the copy module.

Note that when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

When a class itself is pickled, only its name is pickled — the class definition is not pickled, but re-imported by the unpickling process. Therefore, the restriction that the class must be defined at the top level in a module applies to pickled classes as well.

The interface can be summarized as follows.

To pickle an object `x` onto a file `f`, open for writing:

```python
p = pickle.Pickler(f)
p.dump(x)
```

A shorthand for this is:

```python
pickle.dump(x, f)
```

To unpickle an object `x` from a file `f`, open for reading:

```python
u = pickle.Unpickler(f)
x = u.load()
```

A shorthand is:

```python
x = pickle.load(f)
```

The `Pickler` class only calls the method `f.write` with a string argument. The `Unpickler` class calls the methods `f.load` (with an integer argument) and `f.readline` (without argument), both returning a string. It is explicitly allowed to pass non-file objects here, as long as they have the right methods.

The following types can be pickled:

- `None`
- integers, long integers, floating point numbers
- strings
- tuples, lists and dictionaries containing only picklable objects
- classes that are defined at the top level in a module
- instances of such classes whose `__dict__` or `__setstate__()` is picklable

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have been written to the file.

It is possible to make multiple calls to the `dump()` method of the same `Pickler` instance. These must then be matched to the same number of calls to the `load()` instance of the corresponding `Unpickler` instance. If the same object is pickled by multiple `dump()` calls, the `load()` will all yield references to the same object. Warning: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same `Pickler` instance, the object is not pickled again — a reference to it is pickled and the `Unpickler` will return the old value, not the modified one. (There are two problems here: (a) detecting changes, and (b) marshalling a minimal set of changes. I have no answers. Garbage Collection may also become a problem here.)

Apart from the `Pickler` and `Unpickler` classes, the module defines the following functions, and an exception:

- `dump(object, file)`
  - Write a pickled representation of `object` to the open file object `file`. This is equivalent to `Pickler(file).dump(object)`.
- `load(file)`
  - Read a pickled object from the open file object `file`. This is equivalent to `Unpickler(file).load()`.
- `dumps(object)`
  - Return the pickled representation of the object as a string, instead of writing it to a file.
- `loads(string)`
  - Read a pickled object from a string instead of a file. Characters in the string past the pickled object's representation are ignored.
3.5 Standard Module *shelve*

A "shelf" is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the pickle module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

To summarize the interface (*key* is a string, *data* is an arbitrary object):

```
import shelve

d = shelve.open(filename) # open, with (g)dbm filename -- no suffix

d[key] = data  # store data at key (overwrites old data if # using an existing key)
data = d[key]  # retrieve data at key (raises KeyError if no # such key)
del d[key]  # delete data stored at key (raises KeyError # if no such key)
flag = d.has_key(key)  # true if the key exists
list = d.keys()  # a list of all existing keys (slow!)
d.close()  # close it
```

Restrictions:

- The choice of which database package will be used (e.g. dbm or gdbm) depends on which interface is available. Therefore it isn’t safe to open the database directly using dbm. The database is also (unfortunately) subject to the limitations of dbm, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- Dependent on the implementation, closing a persistent dictionary may or may not be necessary to flush changes to disk.
- The *shelve* module does not support concurrent read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. UNIX file locking can be used to solve this, but this differs across UNIX versions and requires knowledge about the database implementation used.

3.6 Standard Module *copy*

This module provides generic (shallow and deep) copying operations.

Interface summary:

```
import copy

x = copy.copy(y)  # make a shallow copy of y
x = copy.deepcopy(y)  # make a deep copy of y
```

For module specific errors, *copy.error* is raised.

The difference between shallow and deepcopying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A **shallow copy** constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A **deep copy** constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Two problems often exist with deep copy operations that don’t exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, e.g. administrative data structures that should be shared even between copies.

Python’s *deepcopy()* operation avoids these problems by:

- keeping a table of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This version does not copy types like module, class, function, method, nor stack trace, stack frame, nor file, socket, window, nor array, nor any similar types.

Classes can use the same interfaces to control copying that they use to control pickling: they can define methods called *__getinitargs__()* , *__getstate__()* and *__setstate__()* . See the description of module *pickle* for information on these methods.

3.7 Built-in Module *marshal*

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python
value to a file on a PC, transport the file to a Sun, and read it back there). Details of the format are 
undocumented on purpose; it may change between Python versions (although it rarely does).¹ 

This is not a general "persistence" module. For general persistence and transfer of Python objects 
through RPC calls, see the modules pickle and marshal. The marshal module exists mainly to 
support reading and writing the "pseudo-compiled" code for Python modules of `.pyc` files. 

Not all Python object types are supported; in general, only objects whose value is independent from 
a particular invocation of Python can be written and read by this module. The following types are 
supported: None, integers, long integers, floating point numbers, strings, tuples, lists, dictionaries, 
and code objects, where it should be understood that tuples, lists and dictionaries are only supported 
as long as the values contained therein are themselves supported; and recursive lists and dictionaries 
should not be written (they will cause infinite loops). 

Caveat: On machines where C's long int type has more than 32 bits (such as the DEC Alpha or 
the HP Precision Architecture), it is possible to create plain Python integers that are longer than 32 
bits. Since the current marshal module uses 32 bits to transfer plain Python integers, such values 
are silently truncated. This particularly affects the use of very long integer literals in Python modules 
— these will be accepted by the parser on such machines, but will be silently be truncated when the 
module is read from the .pyc instead.² 

There are functions that read/write files as well as functions operating on strings. 
The module defines these functions: 

dump(value, file) 
Write the value on the open file. The value must be a supported type. The file must be an open 
file object such as sys.stdout or returned by open() or posix.popen(). 

If the value has an unsupported type, garbage is written which cannot be read back by load(). 

load(file) 
Read one value from the open file and return it. If no valid value is read, raise EOFError, 
ValueError or TypeError. The file must be an open file object. 

dumps(value) 
Return the string that would be written to a file by dump(value, file). The value must 
be a supported type. 

loads(string) 
Convert the string to a value. If no valid value is found, raise EOFError, ValueError or 
TypeError. Extra characters in the string are ignored. 

¹The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who 
use the term "marshalling" for shipping of data around in a self-contained form. Strictly speaking, "to marshal" means to 
convert some data from internal to external form (in an RPC buffer for instance) and "unmarshalling" for the reverse process. 
²A solution would be to refuse such literals in the parser, since they are inherently non-portable. Another solution would 
be to let the marshal module raise an exception when an integer value would be truncated. At least one of these solutions 
will be implemented in a future version. 

3.8 Built-in Module `builtins` 

This module provides an interface to the mechanisms used to implement the import statement. It 
defines the following constants and functions:

get_magic() 
Return the magic string value used to recognize byte-compiled code files (".pyc files"). 

get_suffixes() 
Return a list of triples, each describing a particular type of file. Each triple has the form 
(suffix, mode, type), where suffix is a string to be appended to the module name to form 
the filename to search for; mode is the mode string to pass to the built-in open function to 
open the file (this can be 'r' for text files or 'rb' for binary files), and type is the file type, 
which has one of the values PY_SOURCE, PY_COMPILED or C_EXTENSION, defined below. 
(System-dependent values may also be returned.) 

find_module(name, [path]) 
Try to find the module `name` on the search path. By default it looks in sys.path. The return 
value is a triple (`file`, `pathname`, `description`) where `file` is an open file object positioned 
at the beginning, `pathname` is the pathname of the file found, and `description` is a triple as 
contained in the list returned by get_suffixes describing the kind of file found. 

init_builtin(name) 
Initialize the built-in module called `name` and return its module object. If the module was 
already initialized, it will be initialized again. A few modules cannot be initialized twice — 
attempting to initialize these again will raise an ImportError exception. If there is no 
built-in module called `name`, None is returned. 

init_frozen(name) 
Initialize the frozen module called `name` and return its module object. If the module was 
already initialized, it will be initialized again. If there is no frozen module called `name`, 
None is returned. (Frozen modules are modules written in Python whose compiled byte-code 
object is incorporated into a custom-built Python interpreter by Python's freeze utility. See 
Tools/freeze for now.) 

is_builtin(name) 
Return 1 if there is a built-in module called `name` which can be initialized again. Return 0 if 
there is a built-in module called `name` which cannot be initialized again (see init_builtin). 
Return 0 if there is no built-in module called `name`. 

is_frozen(name) 
Return 1 if there is a frozen module (see init_frozen) called `name`, 0 if there is no such 
module. 

load_compiled(name, pathname, [file]) 
Load and initialize a module implemented as a byte-compiled code file and return its module 
object. If the module was already initialized, it will be initialized again. The `file` argument is 
used to create or access a module object. The `pathname` argument points to the byte-compiled 
code file. The optional `file` argument is the byte-compiled code file, open for reading in binary 
mode, from the beginning — if not given, the function opens `pathname`. It must currently 
be a real file object, not a user-defined class emulating a file.

31 32
load_dynamic(name, pathname, [file])

Load and initialize a module implemented as a dynamically loadable shared library and return its module object. If the module was already initialized, it will be initialized again. Some modules don’t like that and may raise an exception. The pathname argument must point to the shared library. The name argument is used to construct the name of the initialization function: an external C function called initname() in the shared library is called. The optional file argument is ignored. (Note: using shared libraries is highly system dependent, and not all systems support it.)

load_source(name, pathname, [file])

Load and initialize a module implemented as a Python source file and return its module object. If the module was already initialized, it will be initialized again. The name argument is used to create or access a module object. The pathname argument points to the source file. The optional file argument is the source file, open for reading as text, from the beginning — if not given, the function opens pathname. It must currently be a real file object, not a user-defined class emulating a file. Note that if a properly matching byte-compiled file (with suffix .pyc) exists, it will be used instead of parsing the given source file.

new_module(name)

Return a new empty module object called name. This object is not inserted in sys.modules.

The following constants with integer values, defined in the module, are used to indicate the search result of imp.find_module.

SEARCH_ERROR
The module was not found.

PY_SOURCE
The module was found as a source file.

PY_COMPILED
The module was found as a compiled code object file.

C_EXTENSION
The module was found as dynamically loadable shared library.

3.8.1 Examples

The following function emulates the default import statement:

```python
import imp
import sys

def __import__ (name, globals= None, locals= None, fromlist= None):
    # Fast path: see if the module has already been imported.
    if sys.modules.has_key(name):
        return sys.modules[name]

    # If any of the following calls raises an exception,
    # there’s a problem we can’t handle -- let the caller handle it.
```

# See if it’s a built-in module.
m = imp.init_builtin(name)
if m:
    return m

# See if it’s a frozen module.
m = imp.init_frozen(name)
if m:
    return m

# Search the default path (i.e. sys.path).
fp, pathname, (suffix, mode, type) = imp.find_module(name)

# See what we got.
# Note that fp will be closed automatically when we return.
if type == imp.C_EXTENSION:
    return imp.load_dynamic(name, pathname)
if type == imp.PY_SOURCE:
    return imp.load_source(name, pathname, fp)
if type == imp.PY_COMPILED:
    return imp.load_compiled(name, pathname, fp)

# Shouldn’t get here at all.
raise ImportError, ‘%s: unknown module type (%d)’ % (name, type)

3.9 Built-in Module __builtins__

This module provides direct access to all ‘built-in’ identifiers of Python, e.g. __builtins__.open is the full name for the built-in function open. See the section on Built-in Functions in the previous chapter.

3.10 Built-in Module __main__

This module represents the (otherwise anonymous) scope in which the interpreter’s main program executes — commands read either from standard input or from a script file.
Chapter 4

String Services

The modules described in this chapter provide a wide range of string manipulation operations. Here’s an overview:

- **string** — Common string operations.
- **regex** — Regular expression search and match operations.
- **regsub** — Substitution and splitting operations that use regular expressions.
- **struct** — Interpret strings as packed binary data.

4.1 Standard Module **string**

This module defines some constants useful for checking character classes and some useful string functions based on regular expressions.

The constants defined in this module are:

- **digits** — The string `'0123456789'`.
- **hexdigits** — The string `'0123456789abcdefABCDEF'`.
- **letters** — The concatenation of the strings **lowercase** and **uppercase** described below.
- **lowercase** — A string containing all the characters that are considered lowercase letters. On most systems this is the string `'abcdefghijklmnopqrstuvwxyz'`. Do not change its definition — the effect on the routines **upper** and **swapcase** is undefined.
- **octdigits** — The string `'01234567'`.
- **uppercase** — A string containing all the characters that are considered uppercase letters. On most systems this is the string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Do not change its definition — the effect on the routines **lower** and **swapcase** is undefined.
- **whitespace** — A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab. Do not change its definition — the effect on the routines **strip** and **split** is undefined.

The functions defined in this module are:

- **atof** — Convert a string to a floating point number. The string must have the standard syntax for a floating point literal in Python, optionally preceded by a sign (`+` or `-`).
- **atoi** — The string `s` to an integer in the given `base`. The string must consist of one or more digits, optionally preceded by a sign (`+` or `-`). The `base` defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): `'0x'` or `'0X'` means 16, `'0'` means 8, anything else means 10. If `base` is 16, a leading `'0x'` or `'0X'` is always accepted. (Note: for a more flexible interpretation of numeric literals, use the built-in function `eval()`.)
- **atol** — The string `s` to a long integer in the given `base`. The string must consist of one or more digits, optionally preceded by a sign (`+` or `-`). The `base` argument has the same meaning as for `atoi()`. A trailing `'l'` or `'L'` is not allowed.
- **expandtabs** — Expand tabs in a string, i.e. replace them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn’t understand other non-printing characters or escape sequences.
- **find** — Return the lowest index in `s` not smaller than `start` where the substring `sub` is found. Return `-1` when `sub` does not occur as a substring of `s` with index at least `start`. If `start` is omitted, it defaults to 0. If `start` is negative, `len(s)` is added.
- **rfind** — Like `find` but find the highest index.
- **index** — Return the index in `s` at which `sub` is found. Raise `ValueError` when the substring is not found.
- **rindex** — Like `rfind` but raise `ValueError` when the substring is not found.
- **count** — Return the number of (non-overlapping) occurrences of substring `sub` in string `s` with index at least `start`. If `start` is omitted, it defaults to 0. If `start` is negative, `len(s)` is added.
Convert letters to lower case.

\texttt{split(s)}  
Return a list of the whitespace-delimited words of the string \texttt{s}.

\texttt{splitfields(s, \textit{sep})}  
Return a list containing the fields of the string \texttt{s}, using the string \textit{sep} as a separator. The list will have one more item than the number of non-overlapping occurrences of the separator in the string. Thus, \texttt{string.splitfields(s, ' ') is not the same as string.split(s).}  
as the latter only returns non-empty words. As a special case, \texttt{splitfields(s, '') returns [s], for any string s. (See also regex.split())}\n
\texttt{join(words)}  
Concatenate a list or tuple of words with intervening spaces.

\texttt{joinfields(words, \textit{sep})}  
Concatenate a list or tuple of words with intervening separators. It is always true that \texttt{string.joinfields(string.splitfields(s, \textit{sep}), \textit{sep}) equals s.}\n
\texttt{strip(s)}  
Remove leading and trailing whitespace from the string \texttt{s}.

\texttt{swapcase(s)}  
Convert lower case letters to upper case and vice versa.

\texttt{upper(s)}  
Convert letters to upper case.

\texttt{ljust(s, width)}  
These functions respectively left-justify, right-justify and center a string in a field of given width. They return a string that is at least \textit{width} characters wide, created by padding the string \texttt{s} with spaces until the given width on the right, left or both sides. The string is never truncated.

\texttt{rjust(s, width)}

\texttt{center(s, width)}

This module is implemented in Python. Much of its functionality has been reimplemented in the built-in module \texttt{strop}. However, you should never import the latter module directly. When \texttt{string} discovers that \texttt{strop} exists, it transparently replaces parts of itself with the implementation from \texttt{strop}. After initialization, there is no overhead in using \texttt{string} instead of \texttt{strop}.

\section{Built-in Module \texttt{regex}}

This module provides regular expression matching operations similar to those found in Emacs. It is always available.

By default the patterns are Emacs-style regular expressions; there is a way to change the syntax to match that of several well-known UNIX utilities.

This module is 8-bit clean: both patterns and strings may contain null bytes and characters whose high bit is set.

\textbf{Please note}: There is a little-known fact about Python string literals which means that you don’t usually have to worry about doubling backslashes, even though they are used to escape special characters in string literals as well as in regular expressions. This is because Python doesn’t remove backslashes from string literals if they are followed by an unrecognized escape character. \textit{However}, if you want to include a literal backslash in a regular expression represented as a string literal, you have to quadruple it. E.g. to extract LaTeX \texttt{\section{...} } headers from a document, you can use this pattern: \texttt{'\\\\section{...}'}.

The module defines these functions, and an exception:

\texttt{match(pattern, string)}  
Return how many characters at the beginning of \texttt{string} match the regular expression \texttt{pattern}. Return -1 if the string does not match the pattern (this is different from a zero-length match!).

\texttt{search(pattern, string)}  
Return the first position in \texttt{string} that matches the regular expression \texttt{pattern}. Return -1 if no position in the string matches the pattern (this is different from a zero-length match anywhere!).

\texttt{compile(pattern [, translate ])}  
Compile a regular expression pattern into a regular expression object, which can be used for matching using its \texttt{match} and \texttt{search} methods, described below. The optional argument \texttt{translate}, if present, must be a 256-character string indicating how characters (both of the pattern and of the strings to be matched) are translated before comparing them; the 1st element of the string gives the translation for the character with ASCII code \texttt{i}. This can be used to implement case-insensitive matching; see the \texttt{casefold} data item below.

The sequence

\texttt{prog = regex.compile(pat)}

\texttt{result = prog.match(str)}

is equivalent to

\texttt{result = regex.match(pat, str)}

but the version using \texttt{compile()} is more efficient when multiple regular expressions are used concurrently in a single program. (The compiled version of the last pattern passed to \texttt{regex.match()} or \texttt{regex.search()} is cached, so programs that use only a single regular expression at a time needn’t worry about compiling regular expressions.)

\texttt{set_syntax(\textit{flags})}  
Set the syntax to be used by future calls to \texttt{compile}, \texttt{match} and \texttt{search}. (Already compiled expression objects are not affected.) The argument is an integer which is the OR of several flag bits. The return value is the previous value of the syntax flags. Names for the flags are defined in the standard module \texttt{regex_syntax}; read the file ‘\texttt{regex_syntax.py}’ for more information.
symcomp (pattern [, translate ])

This is like compile, but supports symbolic group names: if a parenthesis-enclosed group begins with a group name in angular brackets, e.g. `\(\{\text{id}\}\)`\(\{a-z\}[a-z0-9]*\)`
\`, the group can be referenced by its name in arguments to the group method of the resulting compiled regular expression object, like this: `p.group(\'id\')`. Group names may contain alphanumeric characters and `-` only.

error

Exception raised when a string passed to one of the functions here is not a valid regular expression (e.g., unmatched parentheses) or when some other error occurs during compilation or matching. (It is never an error if a string contains no match for a pattern.)

casefold

A string suitable to pass as translate argument to compile to map all upper case characters to their lowercase equivalents.

Compiled regular expression objects support these methods:

match (string [, pos ])

Return how many characters at the beginning of string match the compiled regular expression. Return -1 if the string does not match the pattern (this is different from a zero-length match!).

The optional second parameter pos gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the `''` pattern character matches at the real begin of the string and at positions just after a newline, not necessarily at the index where the search is to start.

search (string [, pos ])

Return the first position in string that matches the regular expression pattern. Return -1 if no position in the string matches the pattern (this is different from a zero-length match anywhere!).

The optional second parameter has the same meaning as for the match method.

group (index, index, ...)

This method is only valid when the last call to the match or search method found a match. It returns one or more groups of the match. If there is a single index argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument.

If the index is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1,99], it is the string matching the corresponding parenthesized group (using the default syntax, groups are parenthesized using `()` and `)`). If no such group exists, the corresponding result is None.

If the regular expression was compiled by symcomp instead of compile, the index arguments may also be strings identifying groups by their group name.

Compiled regular expressions support these data attributes:

regs

When the last call to the match or search method found a match, this is a tuple of pairs of indices corresponding to the beginning and end of all parenthesized groups in the pattern. Indices are relative to the string argument passed to match or search. The 0-th tuple gives the beginning and end of the whole pattern. When the last match or search failed, this is None.

last

When the last call to the match or search method found a match, this is the string argument passed to that method. When the last match or search failed, this is None.

translate

This is the value of the translate argument to regex.compile that created this regular expression object. If the translate argument was omitted in the regex.compile call, this is None.

givenpat

The regular expression pattern as passed to compile or symcomp.

realmat

The regular expression after stripping the group names for regular expressions compiled with symcomp. Same as givenpat otherwise.

groupindex

A dictionary giving the mapping from symbolic group names to numerical group indices for regular expressions compiled with symcomp. None otherwise.

4.3 Standard Module re

This module defines a number of functions useful for working with regular expressions (see built-in module re).

sub (pat, repl, str)

Replace the first occurrence of pattern pat in string str by replacement repl. If the pattern isn’t found, the string is returned unchanged. The pattern may be a string or an already compiled pattern. The replacement may contain references `\d+` to subpatterns and escaped backslashes.

gsub (pat, repl, str)

Replace all (non-overlapping) occurrences of pattern pat in string str by replacement repl. The same rules as for sub() apply. Empty matches for the pattern are replaced only when not adjacent to a previous match, so e.g. gsub(\'', \'', \'abc\') returns \'a-b-c\'.

split (str, pat)

Split the string str in fields separated by delimiters matching the pattern pat, and return a list containing the fields. Only non-empty matches for the pattern are considered, so e.g. split(\'a:b\', \':\'' returns \[\'a\', \'b\'] and split(\'abc\', \'', '') returns \[\'abc\'].

4.4 Built-in Module struct

This module performs conversions between Python values and C structs represented as Python strings. It uses `form` strings (explained below) as compact descriptions of the lay-out of the C structs and the intended conversion to/from Python values.

See also built-in module array.
The module defines the following exception and functions:

- **error**
  Exception raised on various occasions; argument is a string describing what is wrong.

- **pack**(fmt, v1, v2, ...)
  Return a string containing the values v1, v2, ... packed according to the given format. The arguments must match the values required by the format exactly.

- **unpack**(fmt, string)
  Unpack the string (presumably packed by `pack(fmt, ...)` according to the given format). The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (i.e., `len(string)` must equal `calcsize(fmt)`).

- **calcsize**(fmt)
  Return the size of the struct (and hence of the string) corresponding to the given format.

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types:

<table>
<thead>
<tr>
<th>Format</th>
<th>C</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code></td>
<td>pad byte</td>
<td>no value</td>
</tr>
<tr>
<td><code>c</code></td>
<td>char</td>
<td>string</td>
</tr>
<tr>
<td><code>b</code></td>
<td>signed char</td>
<td>integer</td>
</tr>
<tr>
<td><code>h</code></td>
<td>short</td>
<td>integer</td>
</tr>
<tr>
<td><code>i</code></td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td><code>l</code></td>
<td>long</td>
<td>integer</td>
</tr>
<tr>
<td><code>f</code></td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td><code>d</code></td>
<td>double</td>
<td>float</td>
</tr>
</tbody>
</table>

A format character may be preceded by an integral repeat count; e.g., the format string `4h` means exactly the same as `hhhh`, `4f` means exactly the same as `ffff`, etc. The format string must contain exactly the amount of data required by the format (i.e., `len(string)` must equal `calcsize(fmt)`).

The `calcsize()` function, which is not standard C, is not available on all platforms.

More format characters are planned, e.g., `s` for character arrays, uppercase for unsigned variants, and a way to specify the byte order, which is useful for (de)constructing network packets and reading/writing portable binary file formats like TIFF and AIFF.

With this new function, it is possible to build complex structs in a single call, e.g.,

```python
struct = unpack('hhll', string)
```

This is faster than

```python
struct = unpack('h', string)
struct = unpack('l', struct)
struct = unpack('l', struct)
```

Note that `pack` and `unpack` must have a different alignment pattern than their C equivalents; they take a `no padding` approach (there is no such thing in Python).

The module also defines two mathematical constants: `pi` and `e`.

## Chapter 5: Miscellaneous Services

### 5.1 Built-in Module `math`

This module is always available. It provides access to the mathematical functions defined by the C standard. They are:

- `acos(x)`
- `asin(x)`
- `atan(x)`
- `atan2(x, y)`
- `ceil(x)`
- `cos(x)`
- `cosh(x)`
- `exp(x)`
- `fabs(x)`
- `floor(x)`
- `fmod(x, y)`
- `frexp(x)`
- `hypot(x, y)`
- `ldexp(x, y)`
- `log(x)`
- `log10(x)`
- `modf(x)`
- `pow(x, y)`
- `sin(x)`
- `sinh(x)`
- `sqrt(x)`
- `tan(x)`
- `tanh(x)`

Note that `fmod` and most other arithmetic operations have a different alignment pattern than their C equivalents; they take a "no padding" approach (there is no such thing in Python).

The module also defines two mathematical constants: `pi` and `e`.

### 5.2 Standard Module `random`

This module implements a pseudo-random number generator with an interface similar to `rand()` in

- `random.seed([s])`
- `random.random()` returns a random float in the range `[0,1)`
- `random.randint(a, b)` returns an integer in the range `[a, b]`
- `random.randrange(start, stop[, step])` returns a random integer in the range `[start, stop)`
- `random.choice(iterable)` returns a random element from `iterable`
- `random.shuffle(x)` shuffles the sequence `x` in place
- `random.sample(population, k)` returns a list of `k` unique elements chosen from the range `population`

In addition, this module defines the following functions:

- `random.gauss(mu, sigma)` returns a random float following the normal distribution with mean `mu` and standard deviation `sigma`
- `random.poisson(lambd)` returns a random integer following the Poisson distribution with parameter `lambd`

Here's an overview of the functions:

- `random()` returns a random float in the range `[0,1)`
- `random.random()` returns a random float in the range `[0,1)`
- `random.randint(a, b)` returns an integer in the range `[a, b]`
- `random.randrange(start, stop[, step])` returns a random integer in the range `[start, stop)`
- `random.choice(iterable)` returns a random element from `iterable`
- `random.shuffle(x)` shuffles the sequence `x` in place
- `random.sample(population, k)` returns a list of `k` unique elements chosen from the range `population`
- `random.gauss(mu, sigma)` returns a random float following the normal distribution with mean `mu` and standard deviation `sigma`
- `random.poisson(lambd)` returns a random integer following the Poisson distribution with parameter `lambd`

In addition, this module defines the following functions:

- `random()` returns a random float in the range `[0,1)`
- `random.random()` returns a random float in the range `[0,1)`
- `random.randint(a, b)` returns an integer in the range `[a, b]`
- `random.randrange(start, stop[, step])` returns a random integer in the range `[start, stop)`
- `random.choice(iterable)` returns a random element from `iterable`
- `random.shuffle(x)` shuffles the sequence `x` in place
- `random.sample(population, k)` returns a list of `k` unique elements chosen from the range `population`
- `random.gauss(mu, sigma)` returns a random float following the normal distribution with mean `mu` and standard deviation `sigma`
- `random.poisson(lambd)` returns a random integer following the Poisson distribution with parameter `lambd`


**rand()**

- Returns an integer random number in the range [0 ... 32768).

**choice(s)**

- Returns a random element from the sequence (string, tuple or list)s.

**srand(seed)**

- Initializes the random number generator with the given integral seed. When the module is first imported, the random number is initialized with the current time.

### 5.3 Standard Module `whrandom`

This module implements a Wichmann-Hill pseudo-random number generator. It defines the following functions:

**random()**

- Returns the next random floating point number in the range [0.0 ... 1.0).

**seed(x, y, z)**

- Initializes the random number generator from the integers x, y, and z. When the module is first imported, the random number is initialized using values derived from the current time.

### 5.4 Built-in Module `array`

This module defines a new object type which can efficiently represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a *type code*, which is a single character. The following type codes are defined:

<table>
<thead>
<tr>
<th>Typecode</th>
<th>Type</th>
<th>Minimal size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>character</td>
<td>1</td>
</tr>
<tr>
<td>'b'</td>
<td>signed integer</td>
<td>1</td>
</tr>
<tr>
<td>'h'</td>
<td>signed integer</td>
<td>2</td>
</tr>
<tr>
<td>'i'</td>
<td>signed integer</td>
<td>2</td>
</tr>
<tr>
<td>'l'</td>
<td>signed integer</td>
<td>4</td>
</tr>
<tr>
<td>'f'</td>
<td>floating point</td>
<td>4</td>
</tr>
<tr>
<td>'d'</td>
<td>floating point</td>
<td>8</td>
</tr>
</tbody>
</table>

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the *itemsize* attribute.

See also built-in module `struct`.

The module defines the following function:

**array(typecode[, initializer])**

- Return a new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a list or a string. The list or string is passed to the new array's fromlist() or fromstring() method (see below) to add initial items to the array.

Array objects support the following data items and methods:

**typecode**

- The typecode character used to create the array.

**itemsize**

- The length in bytes of one array item in the internal representation.

**append(x)**

- Append a new item with value x to the end of the array.

**byteswap(x)**

- "Byteswap" all items of the array. This is only supported for integer values. It is useful when reading data from a file written on a machine with a different byte order.

**fromfile(f, n)**

- Read n items (as machine values) from the file object f and append them to the end of the array. If less than n items are available, EOFError is raised, but the items that were available are still inserted into the array. f must be a real built-in file object; something else with a read() method won't do.

**fromlist(list)**

- Append items from the list. This is equivalent to for x in list: a.append(x) except that if there is a type error, the array is unchanged.

**fromstring(s)**

- Append items from the string, interpreting the string as an array of machine values (i.e. as if it had been read from a file using the fromfile() method).

**insert(i, x)**

- Insert a new item with value x in the array before position i.

**tofile(f)**

- Write all items (as machine values) to the file object f.

**tolist()**

- Convert the array to an ordinary list with the same items.

**tostring()**

- Convert the array to an ordinary list with the same items.

- Convert the array to an ordinary list with the same items.

- Convert the array to an ordinary list with the same items.

When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a string if the *typecode* is 'c', otherwise it is a list of numbers. The strings are guaranteed to be able to be converted back to an array with the same type and value using reverse quotes (```). Examples:

```python
array('l')
array('c', 'hello world')
array('L', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```
Chapter 6

Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modelled after the Unix or C interfaces but they are available on most other systems as well. Here's an overview:

- **os** — Miscellaneous OS interfaces.
- **time** — Time access and conversions.
- **getopt** — Parser for command line options.
- **tempfile** — Generate temporary file names.

### 6.1 Standard Module os

This module provides a more portable way of using operating system (OS) dependent functionality than importing an OS dependent built-in module like `posix`.

When the optional built-in module `posix` is available, this module exports the same functions and data as `posix`; otherwise, it searches for an OS dependent built-in module like `mac` and exports the same functions and data as found there. The design of all Python's built-in OS dependent modules is such that as long as the same functionality is available, it uses the same interface; e.g., the function `os.stat(file)` returns stat info about a file in a format compatible with the POSIX interface.

Extensions peculiar to a particular OS are also available through the `os` module, but using them is of course a threat to portability!

Note that after the first time `os` is imported, there is no performance penalty in using functions from `os` instead of directly from the OS dependent built-in module, so there should be no reason not to use `os`!

In addition to whatever the correct OS dependent module exports, the following variables and functions are always exported by `os`:

```python
name

The name of the OS dependent module imported. The following names have currently been registered: 'posix', 'nt', 'dos', 'mac'.
```

**path**

The corresponding OS dependent standard module for pathname operations, e.g. `posixpath` or `macpath`. Thus, (given the proper imports), `os.path.split(file)` is equivalent to but more portable than `posixpath.split(file)`.

**curdir**

The constant string used by the OS to refer to the current directory, e.g. ‘.’ for POSIX or ‘:’ for the Mac.

**pardir**

The constant string used by the OS to refer to the parent directory, e.g. ‘..’ for POSIX or ‘::’ for the Mac.

**sep**

The character used by the OS to separate pathname components, e.g. ‘/’ for POSIX or ‘:’ for the Mac. Note that knowing this is not sufficient to be able to parse or concatenate pathnames—better use `os.path.split()` and `os.path.join()`—but it is occasionally useful.

**pathsep**

The character conventionally used by the OS to separate search path components (as in `$PATH`), e.g. ‘:’ for POSIX or ‘;’ for MS-DOS.

**defpath**

The default search path used by `os.exec*p*()` if the environment doesn’t have a ‘PATH’ key.

```python
defcl(path, arg0, arg1, ...)
This is equivalent to `os.execl(path, (arg0, arg1, ...))`.
defcle(path, arg0, arg1, ..., env)
This is equivalent to `os.execl(path, (arg0, arg1, ..., env))`.
defclp(path, arg0, arg1, ...)
This is equivalent to `os.execlp(path, (arg0, arg1, ...))`.
defcvp(path, args)
This is like `os.execlv(path, args)` but duplicates the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from `os.environ['PATH']`.
defcvpe(path, args, env)
This is a cross between `os.execlv()` and `os.execlvp()`. The directory list is obtained from `env['PATH']`.
```

(The functions `os.execv()` and `execve()` are not documented here, since they are implemented by the OS dependent module. If the OS dependent module doesn’t define either of these, the functions that rely on it will raise an exception. They are documented in the section on module `posix`, together with all other functions that `os` imports from the OS dependent module.)
6.2 Built-in Module time

This module provides various time-related functions. It is always available.

An explanation of some terminology and conventions is in order.

- The “epoch” is the point where the time starts. On January 1st of that year, at 0 hours, the “time since the epoch” is zero. For UNIX, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.
- UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time). The acronym UTC is not a mistake but a compromise between English and French.
- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.
- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most UNIX systems, the clock “ticks” only 50 or 100 times a second, and on the Mac, times are only accurate to whole seconds.

The module defines the following functions and data items:

- `altzone`
  - The offset of the local DST timezone, in seconds west of the 0th meridian, if one is defined. Negative if the local DST timezone is east of the 0th meridian (as in Western Europe, including the UK). Only use this if `daylight` is nonzero.
- `asctime(tuple)`
  - Convert a tuple representing a time as returned by `gmtime()` or `localtime()` to a 24-character string of the following form: ‘Sun Jun 20 23:21:05 1993’. Note: unlike the C function of the same name, there is no trailing newline.
- `clock()`
  - Return the current CPU time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of “CPU time”, depends on that of the C function of the same name.
- `ctime(secs)`
  - Convert a time expressed in seconds since the epoch to a string representing local time. `ctime(t)` is equivalent to `asctime(localtime(t))`.
- `daylight`
  - Nonzero if a DST timezone is defined.
- `gmtime(secs)`
  - Like `gmtime` but converts to local time. The dst flag is set to 1 when DST applies to the given time.
- `mktime(tuple)`
  - This is the inverse function of `localtime`. Its argument is the full 9-tuple (since the dst flag is needed). It returns an integer.
- `sleep(secs)`
  - Suspend execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.
- `time()`
  - Return the time as a floating point number expressed in seconds since the epoch, in UTC. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second.
- `timezone`
  - The offset of the local (non-DST) timezone, in seconds west of the 0th meridian (i.e. negative in most of Western Europe, positive in the US, zero in the UK).
- `tzname`
  - A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used.

6.3 Standard Module getopt

This module helps scripts to parse the command line arguments in `sys.argv`. It uses the same conventions as the UNIX `getopt()` function (including the special meanings of arguments of the form ‘-’ and ‘--’). It defines the function `getopt.getopt(args, options)` and the exception `getopt.error`.

The first argument to `getopt()` is the argument list passed to the script with its first element chopped off (i.e., `sys.argv[1:]`). The second argument is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (i.e., the same format that UNIX `getopt()` uses). The return value consists of two elements: the first is a list of option-and-value pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of the first argument). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen (e.g., ‘-x’), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Example:
>>> import getopt, string
>>> args = string.split('-a -b -cfoo -d bar al a2')
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'al', 'a2']
>>> optlist, args = getopt.getopt(args, 'abcd: ')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']

The exception \texttt{getopt.error} = \texttt{'getopt.error'} is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error.

### 6.4 Standard Module \texttt{tempfile}

This module generates temporary file names. It is not UNIX specific, but it may require some help on non-UNIX systems.

Note: the modules does not create temporary files, nor does it automatically remove them when the current process exits or dies.

The module defines a single user-callable function:

\begin{verbatim}
mktemp()
\end{verbatim}

Return a unique temporary filename. This is an absolute pathname of a file that does not exist at the time the call is made. No two calls will return the same filename.

The module uses two global variables that tell it how to construct a temporary name. The caller may assign values to them; by default they are initialized at the first call to \texttt{mktemp()}.

\begin{verbatim}
tempdir
\end{verbatim}

When set to a value other than \texttt{None}, this variable defines the directory in which filenames returned by \texttt{mktemp()} reside. The default is taken from the environment variable \texttt{TMPDIR}; if this is not set, either \texttt{/usr/tmp} is used (on UNIX), or the current working directory (all other systems). No check is made to see whether its value is valid.

\begin{verbatim}
template
\end{verbatim}

When set to a value other than \texttt{None}, this variable defines the prefix of the final component of the filenames returned by \texttt{mktemp()}. A string of decimal digits is added to generate unique filenames. The default is either \texttt{"@pid."} where \texttt{pid} is the current process ID (on UNIX), or \texttt{"tmp"} (all other systems).

Warning: if a UNIX process uses \texttt{mktemp()}, then calls \texttt{fork()} and both parent and child continue to use \texttt{mktemp()}. The processes will generate conflicting temporary names. To resolve this, the child process should assign \texttt{None} to \texttt{template}, to force recomputing the default on the next call to \texttt{mktemp()}.

---

### Chapter 7

#### Optional Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on selected operating systems only. The interfaces are generally modeled after the UNIX or C interfaces but they are available on some other systems as well (e.g. Windows or NT). Here’s an overview:

- \texttt{signal} — Set handlers for asynchronous events.
- \texttt{socket} — Low-level networking interface.
- \texttt{select} — Wait for I/O completion on multiple streams.
- \texttt{thread} — Create multiple threads of control within one namespace.

#### 7.1 Built-in Module \texttt{signal}

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signal handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (i.e. Python uses the BSD style interface).
- There is no way to “block” signals temporarily from critical sections (since this is not supported by all UNIX flavors).
- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the “atomic” instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (e.g. regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.
- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying UNIX system’s semantics regarding interrupted system calls.
- Because the C signal handler always returns, it makes little sense to catch synchronous errors like \texttt{SIGFPE} or \texttt{SIGSEGV}.
- Python installs a small number of signal handlers by default: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as a Python exception), SIGINT is translated into a KeyboardInterrupt exception, and SIGTERM is caught so that necessary cleanup (especially sys.exit()) can be performed before actually terminating. All of these can be overridden.

- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads simultaneously is to always perform signal() operations in the main thread of execution. Any thread can perform a signal(), getsignal(), or pause(); only the main thread can set a new signal handler, and the main thread will be the only thread to receive signals (this is enforced by the Python signal module; even if the underlying thread implementation supports sending signals to individual threads). This means that signals can’t be used as a means of interthread communication. Use locks instead.

The variables defined in the signal module are:

- SIG_DFL
  This is one of two standard signal handling options; it simply performs the default function for the signal. For example, on most systems the default action for SIGQUIT is to dump core and exit, while the default action for SIGCLD is to simply ignore it.

- SIG_IGN
  This is another standard signal handler, which will simply ignore the given signal.

- SIG*
  All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in ‘signal.h’. The UNIX man page for ‘signal’ lists the existing signals (on some systems this is ‘signal(2)’), on others the list is in ‘signal(7)’. Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

- NSIG
  One more than the number of the highest signal number.

The signal module defines the following functions:

- alarm(time)
  If time is non-negative, this function requests that a SIGALRM signal be sent to the process in time seconds. Any previously scheduled alarm is canceled (i.e., only one alarm can be scheduled at a time). The returned value is then the number of seconds remaining before any previously set alarm was to have been delivered. If time is zero, no alarm is scheduled, and any scheduled alarm is canceled. The return value is the number of seconds remaining before a previously scheduled alarm. If the return value is zero, no alarm is currently scheduled. (See the UNIX man page alarm(2).)

- getsignal(signalnum)
  Return the current signal handler for the signal signalnum. The returned value may be a callable Python object, or one of the special values signal.SIG_DFL or None. Here, signal.SIG_DFL means that the signal was previously ignored, signal.SIG_DFL means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

- pause()
  Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. (See the UNIX man page signal(2).)

- signal(signalnum, handler)
  Set the handler for signal signalnum to the function handler. handler can be any callable Python object, or one of the special values signal.SIG_IGN or signal.SIG_DFL. The previous signal handler will be returned (see the description of getsignal() above). (See the UNIX man page signal(2).)

  When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a ValueError exception to be raised.

  The handler is called with two arguments: the signal number and the current stack frame (None or a frame object; see the reference manual for a description of frame objects).

### 7.2 Built-in Module socket

This module provides access to the BSD socket interface. It is available on UNIX systems that support this interface.

For an introduction to socket programming (in C), see the following papers: An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest and An Advanced 4.3BSD Interprocess Communication Tutorial, by Samuel J. Leffler et al, both in the UNIX Programmer’s Manual, Supplementary Documents 1 (sections PS1.7 and PS1.8). The UNIX manual pages for the various socket-related system calls are also a valuable source of information on the details of socket semantics.

The Python interface is a straightforward transliteration of the UNIX system call and library interface for sockets to Python’s object-oriented style: the socket() function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with read() and write() operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

Socket addresses are represented as a single string for the AF_UNIX address family and as a pair (host, port) for the AF_INET address family, where host is a string representing either a hostname in Internet domain notation like ‘daring.cwl.nl’ or an IP address like ‘100.50.200.5’, and port is an integral port number. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error socket.error. Non-blocking mode is supported through the setblocking() method.

The module socket exports the following constants and functions:

- error
  This exception is raised for socket- or address-related errors. The accompanying value is either a string telling what went wrong or a pair (errno, string) representing an error returned by a
system call, similar to the value accompanying posix.error.

AF_UNIX
AF_INET

These constants represent the address (and protocol) families, used for the first argument to socket(). If the AF_UNIX constant is not defined then this protocol is unsupported.

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET

These constants represent the socket types, used for the second argument to socket(). (Only SOCK_STREAM and SOCK_DGRAM appear to be generally useful.)

SO_*
SOMAXCONN
MSG_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the setsockopt and getsockopt methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

gethostbyname(hostname)

Translate a host name to IP address format. The IP address is returned as a string, e.g., '100.50.200.5'. If the host name is an IP address itself it is returned unchanged.

gethostname()

Return a string containing the hostname of the machine where the Python interpreter is currently executing. If you want to know the current machine's IP address, use socket.gethostbyname(socket.gethostname()).

gethostbyaddr(ip_address)

Return a triple (hostname, aliaslist, ipaddrlst) where hostname is the primary host name responding to the given ip_address, aliaslist is a (possibly empty) list of alternative host names for the same address, and ipaddrlst is a list of IP addresses for the same interface on the same host (most likely containing only a single address).

getservbyname(servicename, protocolname)

Translate an Internet service name and protocol name to a port number for that service. The protocol name should be 'tcp' or 'udp'.

socket(family, type[, proto])

Create a new socket using the given address family, socket type and protocol number. The address family should be AF_INET or AF_UNIX. The socket type should be SOCK_STREAM, SOCK_DGRAM or perhaps one of the other 'SOCK_' constants. The protocol number is usually zero and may be omitted in that case.

fromfildesc(fd, family[, type[, proto]])

Build a socket object from an existing file descriptor (an integer as returned by a file object's fileno method). Address family, socket type and protocol number are as for the socket function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (e.g. a server started by the UNIX inet daemon).

7.2.1 Socket Objects

Socket objects have the following methods. Except for makefile() these correspond to Unix system calls applicable to sockets.

accept()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

bind(address)

Bind the socket to address. The socket must not already be bound. (The format of address depends on the address family — see above.)

close()

Close the socket. All future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

connect(address)

Connect to a remote socket at address. (The format of address depends on the address family — see above.)

fileno()

Return the socket's file descriptor (a small integer). This is useful with select.

getservbyname(servicename, protocolname)

Translate an Internet service name and protocol name to a port number for that service. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

getservbyport(port)

Return the service name and protocol name for a given port number. This is useful for finding the service name for a port number on a remote IP socket, for instance. (The format of the address returned depends on the address family — see above.)

getsocketoption(level, optname[, buflen])

Return the value of the given socket option (see the Unix man page getsockopt2). The needed symbolic constants (SO_* etc.) are defined in this module. If buflen is absent, an integer option is assumed and its integer value is returned by the function. If buflen is present, it specifies
the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module struct for a way to decode C structures encoded as strings).

listen(backlog)

Listen for connections made to the socket. The backlog argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

makefile [<mode, bufsiz>]

Return a file object associated with the socket. (File objects were described earlier under Built-In Types.) The file object references a dupped version of the socket file descriptor, so the file object and socket object may be closed or garbage-collected independently. The optional mode and bufsize arguments are interpreted the same way as by the built-in open() function.

recv(bufsize [, flags])

Receive data from the socket. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by bufsize. See the Unix manual page for the meaning of the optional argument flags; it defaults to zero.

recvfrom(bufsize [, flags])

Receive data from the socket. The return value is a pair (string, address) where string is a string representing the data received and address is the address of the socket sending the data. The optional flags argument has the same meaning as for recv() above. (The format of address depends on the address family — see above.)

send(string [, flags])

Send data to the socket. The socket must be connected to a remote socket. The optional flags argument has the same meaning as for recv() above. Return the number of bytes sent.

sendto(string [, flags], address)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address. The optional flags argument has the same meaning as for recv() above. Return the number of bytes sent. (The format of address depends on the address family — see above.)

setblocking(flag)

Set blocking or non-blocking mode of the socket: if flag is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a recv call doesn't find any data, or if a send call can't immediately dispose of the data, a socket.error exception is raised; in blocking mode, the calls block until they can proceed.

setsockopt(level, optname, value)

Set the value of the given socket option (see the Unix manual page setsockopt(2)). The needed symbolic constants are defined in the socket module (SO_* etc.). The value can be an integer or a string representing a buffer. In the latter case it is up to the caller to ensure that the string contains the proper bits (see the optional built-in module struct for a way to encode C structures as strings).

shutdown(how)

Shut down one or both halves of the connection. If how is 0, further receives are disallowed. If how is 1, further sends are disallowed. If how is 2, further sends and receives are disallowed.

Note that there are no methods read() or write(); use recv() and send() without flags argument instead.

7.2.2 Example

Here are two minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence socket, bind, listen, accept (possibly repeating the accept to service more than one client), while a client only needs the sequence socket, connect. Also note that the server does not send/receive on the socket it is listening on but on the new socket returned by accept.

```
# Echo server program from socket import *
HOST = '' # Symbolic name meaning the local host
PORT = 50007 # Arbitrary non-privileged server
s = socket(AF_INET, SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
while 1:
data = conn.recv(1024)
if not data: break
conn.send(data)
conn.close()
```

```
# Echo client program from socket import *
HOST = 'daring.cwi.nl' # The remote host
PORT = 50007 # The same port as used by the server
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
data = s.recv(1024)
s.send('Hello, world!')
data = s.recv(1024)
s.close()
print 'Received', `data`
error
The exception raised when an error occurs. The accompanying value is a pair containing the numeric error code from \texttt{errno} and the corresponding string, as would be printed by the C function \texttt{perror()}.

\texttt{select(iwtd, owtd, ewtd[, timeout])}
This is a straightforward interface to the UNIX \texttt{select()} system call. The first three arguments are lists of `waitable objects': either integers representing UNIX file descriptors or objects with a parameterless method named \texttt{fileno()} returning such an integer. The three lists of waitable objects are for input, output and `exceptional conditions', respectively. Empty lists are allowed.

The optional \texttt{timeout} argument specifies a time-out as a floating point number in seconds. When the \texttt{timeout} argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Amongst the acceptable object types in the lists are Python file objects (e.g. \texttt{sys.stdin}, or objects returned by \texttt{open()} or \texttt{posix.popen()}), socket objects returned by \texttt{socket.socket()}, and the module \texttt{stdwin} which happens to define a function \texttt{fileno()} for just this purpose. You may also define a wrapper class yourself, as long as it has an appropriate \texttt{fileno()} method (that really returns a UNIX file descriptor, not just a random integer).

7.4 Built-in Module thread
This module provides low-level primitives for working with multiple threads (a.k.a. light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (a.k.a. mutexes or binary semaphores) are provided.

The module is optional and supported on SGI IRIX 4.x and 5.x and Sun Solaris 2.x systems, as well as on systems that have a PTHREAD implementation (e.g. KSR).

It defines the following constant and functions:

\texttt{error}
Raised on thread-specific errors.

\texttt{start\_new\_thread(func, arg)}
Start a new thread. The thread executes the function \texttt{func} with the argument list \texttt{arg} (which must be a tuple). When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

\texttt{exit()} This is a shorthand for thread.exit\_thread().

\texttt{exit\_thread()} Raise the SystemExit exception. When not caught, this will cause the thread to exit silently.

\texttt{allocate\_lock()} Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

\texttt{get\_ident()} Return the `thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

Lock objects have the following methods:

\texttt{acquire([wait\_lag])}
Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that's their reason for existence), and returns \texttt{None}. If the integer \texttt{wait\_lag} argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as before. If an argument is present, the return value is \texttt{1} if the lock is acquired successfully, \texttt{0} if not.

\texttt{release()} Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

\texttt{locked()} Return the status of the lock: \texttt{1} if it has been acquired by some thread, \texttt{0} if not.

Caveats:

- Threads interact strangely with interrupts: the \texttt{KeyboardInterrupt} exception will be received by an arbitrary thread. (When the \texttt{signal} module is available, interrupts always go to the main thread.)

- Calling \texttt{sys.exit()} or raising the \texttt{SystemExit} is equivalent to calling \texttt{thread.exit\_thread()}.

- Not all built-in functions that may block waiting for I/O allow other threads to run. (The most popular ones (\texttt{sleep, read, select}) work as expected.)
Chapter 8

UNIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to the UNIX operating system, or in some cases to some or many variants of it. Here’s an overview:

- **posix** — The most common Posix system calls (normally used via module os).
- **posixpath** — Common Posix pathname manipulations (normally used via os.path).
- **pwd** — The password database (getpwnam() and friends).
- **grp** — The group database (getgrnam() and friends).
- **dbm** — The standard “database” interface, based on ndbm.
- **gdbm** — GNU’s reinterpretation of dbm.
- **termios** — Posix style tty control.
- **fcntl** — The fcntl() and ioctl() system calls.
- **posixfile** — A file-like object with support for locking.

8.1 Built-in Module posix

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised UNIX interface).

Do not import this module directly. Instead, import the module os, which provides a portable version of this interface. On UNIX, the os module provides a superset of the posix interface. On non-UNIX operating systems the posix module is not available, but a subset is always available through the os interface. Once os is imported, there is no performance penalty in using it instead of posix.

The descriptions below are very terse; refer to the corresponding UNIX manual entry for more information. Arguments called path refer to a pathname given as a string.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise posix.error, described below.

Module posix defines the following data items:

- **environ**
  A dictionary representing the string environment at the time the interpreter was started. For example, posix.environ["HOME"] is the pathname of your home directory, equivalent to geteuid("HOME") in C. Modifying this dictionary does not affect the string environment passed on by execv(), popen() or system(); if you need to change the environment, pass environ to execv() or add variable assignments and export statements to the command string for system() or popen().

- **error**
  This exception is raised when a POSIX function returns a POSIX-related error (e.g., not for illegal argument types). Its string value is 'posix.error'. The accompanying value is a pair containing the numeric error code from errno and the corresponding string, as would be printed by the C function perror().

It defines the following functions and constants:

- **chdir**(path)
  Change the current working directory to path.

- **chmod**(path, mode)
  Change the mode of path to the numeric mode.

- **chown**(path, uid, gid)
  Change the owner and group id of path to the numeric uid and gid. (Not on MS-DOS.)

- **close**(fd)
  Close file descriptor fd.
  Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by posix.open() or posix.pipe(). To close a “file object” returned by the built-in function open or by posix.popen or posix.fdopen, use its close() method.

- **dup**(fd)
  Return a duplicate of file descriptor fd.

- **dup2**(fd1, fd2)
  Duplicate file descriptor fd1 to fd2, closing the latter first if necessary. Return None.

- **execv**(path, args)
  Execute the executable path with argument list args, replacing the current process (i.e., the Python interpreter). The argument list may be a tuple or list of strings. (Not on MS-DOS.)

- **execve**(path, args, env)
  Execute the executable path with argument list args, and environment env, replacing the current process (i.e., the Python interpreter). The argument list may be a tuple or list of strings. The environment must be a dictionary mapping strings to strings. (Not on MS-DOS.)

- **exit**(n)

1The problem with automatically passing environ is that there is no portable way of changing the environment.
Exit to the system with status n, without calling cleanup handlers, flushing stdio buffers, etc.

(Not on MS-DOS.)

Note: the standard way to exit is sys.exit(n). posix._exit() should normally only
be used in the child process after a fork().

fopen(fd, [mode[, bufsize]])

Return an open file object connected to the file descriptor fd. The mode and bufsize arguments
have the same meaning as the corresponding arguments to the built-in open() function.

fork()

Fork a child process. Return 0 in the child, the child's process id in the parent. (Not on
MS-DOS.)

fstat(fd)

Return status for file descriptor fd, like stat().

getcwd()

Return a string representing the current working directory.

getegid()

Return the current process's effective group id. (Not on MS-DOS.)

geteuid()

Return the current process's effective user id. (Not on MS-DOS.)

getgid()

Return the current process's group id. (Not on MS-DOS.)

getgid()

Return the current process's group id. (Not on MS-DOS.)

getpgrp()

Return the parent's process id. (Not on MS-DOS.)

getppid()

Return the current process's user id. (Not on MS-DOS.)

kill(pid, sig)

Kill the process pid with signal sig. (Not on MS-DOS.)

link(src, dst)

Create a hard link pointing to src named dst. (Not on MS-DOS.)

listdir(path)

Return a list containing the names of the entries in the directory. The list is in arbitrary order.
It includes the special entries '.' and '..' if they are present in the directory.

lseek(fd, pos, how)

Set the current position of file descriptor fd to position pos, modified by how: 0 to set the
position relative to the beginning of the file; 1 to set it relative to the current position; 2 to set it
relative to the end of the file.

lstat(path)

Like stat(), but do not follow symbolic links. (On systems without symbolic links, this is
identical to posix.stat.)

mkdir(path, mode)

Create a directory named path with numeric mode mode.

nice(increment)

Add incr to the process' "niceness". Return the new niceness. (Not on MS-DOS.)

open(file, flags, mode)

Open the file file and set various flags according to flags and possibly its mode according to
mode. Return the file descriptor for the newly opened file.

Note: this function is intended for low-level I/O. For normal usage, use the built-in function
open, which returns a "file object" with read() and write() methods (and many more).

pipe()

Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing,
respectively. (Not on MS-DOS.)

popen(command[, mode[, bufsize]])

Open a pipe to or from command. The return value is an open file object connected to the
pipe, which can be read or written depending on whether mode is 'r' (default) or 'w'. The
bufsize argument has the same meaning as the corresponding argument to the built-in open() function.

(Not on MS-DOS.)

read(fd, n)

Read at most n bytes from file descriptor fd. Return a string containing the bytes read.

Note: this function is intended for low-level I/O and must be applied to a file descriptor as
returned by posix.open() or posix.pipe(). To read a "file object" returned by the
built-in function open or by posix.popen or posix.fdopen, or sys.stdin, use its
read() or readline() methods.

readlink(path)

Return a string representing the path to which the symbolic link points. (On systems without
symbolic links, this always raises posix.error.)

rename(src, dst)

Rename the file or directory src to dst.

rmdir(path)

Remove the directory path.

setgid(gid)

Set the current process's group id. (Not on MS-DOS.)

setuid(uid)

Set the current process's user id. (Not on MS-DOS.)

stat(path)

Perform a stat system call on the given path. The return value is a tuple of at least 10 integers
giving the most important (and portable) members of the stat structure, in the order st_mode,
st_ino, st_dev, st_nlink, st_uid, st_gid, st_size, st_atime, st_mtime, st_ctime. More items may be added at the end by some implementations. (On MS-DOS,
some items are filled with dummy values.)

Note: The standard module stat defines functions and constants that are useful for extracting
information from a stat structure.

**symlink** (src, dst)
Create a symbolic link pointing to src named dst. (On systems without symbolic links, this always raises posix.error.)

**system**(command)
Execute the command (a string) in a subshell. This is implemented by calling the Standard C function system(), and has the same limitations. Changes to posix.envirom, sys.stdin etc. are not reflected in the environment of the executed command. The return value is the exit status of the process as returned by Standard C's system().

**times**()
Return a 4-tuple of floating point numbers indicating accumulated CPU times, in seconds. The items are: user time, system time, children's user time, and children's system time, in that order. See the UNIX manual page times(2). (Not on MS-DOS.)

**umask**(mask)
Set the current numeric umask and returns the previous umask. (Not on MS-DOS.)

**uname**()
Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: (sysname, nodename, release, version, machine). Some systems truncate the nodename to 8 characters or to the leading component; a better way to get the hostname is socket.gethostname(). (Not on MS-DOS, nor on older UNIX systems.)

**unlink**(path)
Unlink path.

**utime**(path, (atime, mtime))
Set the access and modified time of the file to the given values. (The second argument is a tuple of two items.)

**wait**()
Wait for completion of a child process, and return a tuple containing its pid and exit status indication (encoded as by UNIX). (Not on MS-DOS.)

**waitpid**(pid, options)
Wait for completion of a child process given by pid, and return a tuple containing its pid and exit status indication (encoded as by UNIX). The semantics of the call are affected by the value of the integer options, which should be 0 for normal operation. (If the system does not support waitpid(), this always raises posix.error. Not on MS-DOS.)

**write**(fd, str)
Write the string str to file descriptor fd. Return the number of bytes actually written.

**WHOCHANG**
The option for waitpid() to avoid hanging if no child process status is available immediately.

### 8.2 Standard Module **posixpath**

This module implements some useful functions on POSIX pathnames.

**Do not import this module directly.** Instead, import the module os and use os.path.

**basename**(p)
Return the base name of pathname p. This is the second half of the pair returned by posixpath.split(p).

**commonprefix**(list)
Return the longest string that is a prefix of all strings in list. If list is empty, return the empty string ('').

**exists**(p)
Return true if p refers to an existing path.

**expanduser**(p)
Return the argument with an initial component of `~' or `user' replaced by that user's home directory. An initial `~' is replaced by the environment variable $HOME; an initial `user' is looked up in the password directory through the built-in module pwd. If the expansion fails, or if the path does not begin with a tilde, the path is returned unchanged.

**expandvars**(p)
Return the argument with environment variables expanded. Substrings of the form `%(name)' or `%(name)' are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.

**isabs**(p)
Return true if p is an absolute pathname (begins with a slash).

**isfile**(p)
Return true if p is an existing regular file. This follows symbolic links, so both islink() and isfile() can be true for the same path.

**isdir**(p)
Return true if p is an existing directory. This follows symbolic links, so both islink() and isdir() can be true for the same path.

**islink**(p)
Return true if p refers to a directory entry that is a symbolic link. Always false if symbolic links are not supported.

**ismount**(p)
Return true if pathname p is a mount point: a point in a file system where a different file system has been mounted. The function checks whether p's parent, `/p/..', is on a different device than p, or whether `/p/..' and p point to the same i-node on the same device — this should detect mount points for all UNIX and POSIX variants.

**join**(p, q)
Join the paths p and q intelligently: If q is an absolute path, the return value is q. Otherwise, the concatenation of p and q is returned, with a slash (`/') inserted unless p is empty or ends in a slash.
8.4 Built-in Module `grp`

This module provides access to the UNIX group database. It is available on all UNIX versions.

Group database entries are reported as 4-tuples containing the following items from the group database (see `<grp.h>`), in order: `gr_name`, `gr_passwd`, `gr_gid`, `gr_mem`. The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database.) An exception is raised if the entry asked for cannot be found.

It defines the following items:

- `getgrgid(gid)`
  - Return the group database entry for the given numeric group ID.

- `getgrnam(name)`
  - Return the group database entry for the given group name.

- `getgrall()`
  - Return a list of all available group entries, in arbitrary order.

8.5 Built-in Module `dbm`

Dbm provides python programs with an interface to the unix `dbm` database library. Dbm objects are of the mapping type, so they can be handled just like objects of the built-in `dictionary` type, except that keys and values are always strings, and printing a dbm object doesn’t print the keys and values.

The module defines the following constant and functions:

- `error`
  - Raised on dbm-specific errors, such as I/O errors. `PermissionError` is raised for general mapping errors like specifying an incorrect key.

- `open(filename, rwmode, filemode)`
  - Open a dbm database and return a mapping object. `filename` is the name of the database file (without the `.dir` or `.pag` extensions), `rwmode` is `'r'`, `'w'` or `'c'` to open the database for reading, writing or both respectively, and `filemode` is the UNIX mode of the file, used only when the database has to be created (but to be supplied at all times).

8.6 Built-in Module `gdbm`

Gdbm provides python programs with an interface to the GNU `gdbm` database library. Gdbm objects are of the mapping type, so they can be handled just like objects of the built-in `dictionary` type, except that keys and values are always strings, and printing a gdbm object doesn’t print the keys and values.

The module is based on the Dbm module, modified to use GDBM instead.

The module defines the following constant and functions:

- `error`
Raised on gdbm-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

```python
def open(filename, rwmode, filename)
    filename is the name of the database file, rwmode is 'r', 'w', 'c', or 'n' for reader, writer (this also gives read access), create(writer, writer, but create the database if it doesn't already exist) and newdb (which will always create a new database). Only one writer may open a gdbm file and many readers may open the file. Readers and writers cannot open the gdbm file at the same time. Note that the `GDBM_FAST` mode of opening the database is not supported.
```

8.7 Built-in Module termios

This module provides an interface to the Posix calls for tty I/O control. For a complete description of these calls, see the Posix or Unix manual pages. It is only available for those Unix versions that support Posix `termios` style tty I/O control (and then only if configured at installation time).

All functions in this module take a file descriptor `fd` as their first argument. This must be an integer file descriptor, such as returned by `os.stdin.fileno()`.

This module should be used in conjunction with the `TERMIOS` module, which defines the relevant symbolic constants (see the next section).

The module defines the following functions:

- `tcgetattr(fd)`
  - Return a list containing the tty attributes for file descriptor `fd`, as follows: `[iflag, oflag, cflag, ispeed, ospeed, cc]` where `cc` is a list of the tty special characters (each a string of length 1, except the items with indices `VMIN` and `VTIME`, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the `cc` array must be done using the symbolic constants defined in the `TERMIOS` module.

- `tcsetattr(fd, when, attributes)`
  - Set the tty attributes for file descriptor `fd` from the `attributes`, which is a list like the one returned by `tcgetattr()`. The `when` argument determines when the attributes are changed: `TERMIOS.TCSANOW` to change immediately, `TERMIOS.TCSADRAIN` to change after transmitting all queued input or output, or `TERMIOS.TCIOFLUSH` to change after transmitting all queued output and discarding all queued input.

- `tcsendbreak(fd, duration)`
  - Send a break on file descriptor `fd`. A zero `duration` sends a break for 0.25–0.5 seconds; a nonzero `duration` has a system dependent meaning.

- `tcdrain(fd)`
  - Wait until all output written to file descriptor `fd` has been transmitted.

- `tcflush(fd, queue)`
  - Discard queued data on file descriptor `fd`. The `queue` selector specifies which queue: `TERMIOS.TCIFLUSH` for the input queue, `TERMIOS.TCIOFLUSH` for the output queue, or `TERMIOS.TCIOFLUSH` for both queues.

8.7.1 Example

Here's a function that prompts for a password with echoing turned off. Note the technique using a separate `termios.tcgetattr()` call and a `try ... finally` statement to ensure that the old tty attributes are restored exactly no matter what happens:

```python
def getpass(prompt = "Password: "):
    import termios, TERMIOS, sys
    fd = sys.stdin.fileno() #filename
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    try:
        termios.tcgetattr(fd, TERMIOS.TCSADRAIN, new)
        passwd = raw_input(prompt)
        finally:
            termios.tcgetattr(fd, TERMIOS.TCSADRAIN, old)
    return passwd
```

8.8 Standard Module TERMIOS

This module defines the symbolic constants required to use the `termios` module (see the previous section). See the Posix or Unix manual pages (or the source) for a list of those constants.

Note: this module resides in a system-dependent subdirectory of the Python library directory. You may have to generate it for your particular system using the script `tools/scripts/h2py.py`.

8.9 Built-in Module fcntl

This module performs file control and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. File descriptors can be obtained with the `file() method of a file or socket object.

The module defines the following functions:

- `fcntl(fd, op, arg)`
  - Perform the requested operation on file descriptor `fd`. The operation is defined by `op` and is operating system dependent. Typically these codes can be retrieved from the library module `FCNTL`. The argument `arg` is optional, and defaults to the integer value 0. When it is present,
it can either be an integer value, or a string. With the argument missing or an integer value, the 
return value of this function is the integer return value of the real `fcntl()` call. When the 
argument is a string it represents a binary structure, e.g. created by `struct.pack()`. The 
binary data is copied to a buffer whose address is passed to the real `fcntl()` call. The return 
value after a successful call is the contents of the buffer, converted to a string object. In case 
the `fcntl()` fails, an IOError will be raised.

`fcntl(fd, op, arg)`
This function is identical to the `fcntl()` function, except that the operations are typically 
defined in the library module `IOCTL`. If the library modules `FCNTL` or `IOCTL` are missing, you can find the opcodes in the C include files `sys/fcntl.h` and `sys/ioctl.h`. You can create the modules yourself with the `h2py` script, found in the `Demo/scripts` directory.

Examples (all on a SVR4 compliant system):

```python
import struct, FCNTL

file = open(...)
rv = fcntl(file.fileno(), FCNTL.O_NDELAY, 1)
lockdata = struct.pack('hhllhh', FCNTL.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl(file.fileno(), FCNTL.F_SETLK, lockdata)
```

Note that in the first example the return value variable `rv` will hold an integer value; in the second example it will hold a string value.

### 8.10 Standard Module posixfile

This module implements some additional functionality over the built-in file objects. In particular, it implements file locking, control over the file flags, and an easy interface to duplicate the file object. The module defines a new file object, the posixfile object. It has all the standard file object methods and adds the methods described below. This module only works for certain flavors of UNIX, since it uses `fcntl()` for file locking.

To instantiate a posixfile object, use the `open()` function in the posixfile module. The resulting object looks and feels roughly the same as a standard file object.

The posixfile module defines the following constants:

- **SEEK_SET**:Offset is calculated from the start of the file
- **SEEK_CUR**: Offset is calculated from the current position in the file
- **SEEK_END**: Offset is calculated from the end of the file

The posixfile module defines the following functions:

- `open(filename [, mode [, bufsize]])`:
  Create a new posixfile object with the given filename and mode. The `filename, mode and bufsize` arguments are interpreted the same way as by the built-in `open()` function.

- `fileopen(fileobject)`:
  Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

- `lock(len [, start [, whence]])`:
  Lock the specified section of the file that the file object is referring to. The format is explained below in a table. The `len` argument specifies the length of the section that should be locked. The default is 0. `start` specifies the starting offset of the section, where the default is 0. The `whence` argument specifies where the offset is relative to. It accepts one of the constants SEEK_SET, SEEK_CUR or SEEK_END. The default is SEEK_SET. For more information about the arguments refer to the `fcntl` manual page on your system.

  ```python
  def lock(self, len, start=0, whence=SEEK_SET):
      pass
  ```

- `flags( [flags] )`:
  Set the specified flags for the file that the file object is referring to. The new flags are ORed with the old flags, unless specified otherwise. The format is explained below in a table. Without the `flags` argument a string indicating the current flags is returned (this is the same as the '?' modifier). For more information about the flags refer to the `fcntl` manual page on your system.

  ```python
  def flags(self, flags=None):
      pass
  ```

- `dup()`:
  Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

  ```python
  def dup(self):
      pass
  ```

- `dup2(fd)`:
  Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

  ```python
  def dup2(self, fd):
      pass
  ```

- `file()`:
  Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

  ```python
  def file(self):
      pass
  ```

All methods return IOError when the request fails.

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'?'</td>
<td>unlock the specified region</td>
</tr>
<tr>
<td>'r'</td>
<td>request a read lock for the specified section</td>
</tr>
<tr>
<td>'w'</td>
<td>request a write lock for the specified section</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:
<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>`'!'</td>
<td>return the first lock conflicting with the requested lock, or None if there is no conflict.</td>
<td>(1)</td>
</tr>
<tr>
<td>`'?'</td>
<td>wait until the lock has been granted</td>
<td></td>
</tr>
</tbody>
</table>

Note:

(1) The lock returned is in the format (mode, len, start, whence, pid) where mode is a character representing the type of lock ('r' or 'w'). This modifier prevents a request from being granted; it is for query purposes only.

Format character for the flags() method have the following meaning:

<table>
<thead>
<tr>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>append only flag</td>
</tr>
<tr>
<td>'c'</td>
<td>close on exec flag</td>
</tr>
<tr>
<td>'n'</td>
<td>no-delay flag (also called non-blocking flag)</td>
</tr>
<tr>
<td>'s'</td>
<td>synchronization flag</td>
</tr>
</tbody>
</table>

In addition the following modifiers can be added to the format:

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>'!='</td>
<td>turn the specified flags off, instead of the default 'on'</td>
<td>(1)</td>
</tr>
<tr>
<td>'w='</td>
<td>replace the flags, instead of the default 'OR' operation</td>
<td>(1)</td>
</tr>
<tr>
<td>'?='</td>
<td>return a string in which the characters represent the flags that are set.</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Note:

(1) The ! and = modifiers are mutually exclusive.

(2) This string represents the flags after they may have been altered by the same call.

Examples:

```python
from posixfile import *

file = open('/tmp/test', 'w')
file.lock('w|')
...
file.lock('u')
file.close()
```

**Chapter 9**

**The Python Debugger**

The module `pdb` defines an interactive source code debugger for Python programs. It supports setting breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible — it is actually defined as a class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the (also undocumented) modules `bdb` and `cmd`.

A primitive windowing version of the debugger also exists — this is module `wdb`, which requires STDWIN (see the chapter on STDWIN specific modules).

The debugger’s prompt is “(Pdb)”. Typical usage to run a program under control of the debugger is:

```python
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
<string>(0)?()
(Pdb) continue
<string>(1)?()
(Pdb) continue
NameError: 'spam'
<string>(1)?()
(Pdb)
```

Typical usage to inspect a crashed program is:

```python
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "/mymodule.py", line 4, in test
```
The module defines the following functions, each enters the debugger in a slightly different way:

**run** *(statement[], globals[, locals] [])*

Execute the statement (given as a string) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type `continue` or you can step through the statement using `step` or `next` (all these commands are explained below). The optional `globals` and `locals` arguments specify the environment in which the code is executed; by default the dictionary of the module `__main__` is used. (See the explanation of the `exec` statement or the `eval` built-in function.)

**runeval** *(expression[, globals[, locals] []]*

Evaluate the expression (given as a a string) under debugger control. When `runeval()` returns, it returns the value of the expression. Otherwise this function is similar to `run()`.

**runcall** *(function[, arguments,...] [])*

Call the function (a function or method object, not a string) with the given arguments. When `runcall()` returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

**set_trace()**

Enter the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

**post_mortem(traceback)**

Enter post-mortem debugging of the given traceback object.

**pm()**

Enter post-mortem debugging of the traceback found in `sys.last_traceback`.

### 9.1 Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. `h (help)` means that either `h` or `help` can be used to enter the help command (but not `he` or `help` or `Hel` or `HELP`). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets (`[]`) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (`|`).

Entering a blank line repeats the last command entered. Exception: if the last command was a “list” command, the next 11 lines are listed.

Commands that the debugger doesn’t recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (`!`). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger’s state is not changed.

**help** *(command [])*

Without argument, print the list of available commands. With a command as argument, print help about that command. “help pdb” displays the full documentation file; if the environment variable `PAGER` is defined, the file is piped through that command instead. Since the command argument must be an identifier, “help exec” must be entered to get help on the “exec” command.

**where**

Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

**up**

Move the current frame one level up in the stack trace (to a newer frame).

**down**

Move the current frame one level down in the stack trace (to an older frame).

**run** *(expression[])*

Evaluate the expression in the current context and print its value. (Note: `print` can also be used, but is not a debugger command — this executes the Python `print` statement.)
Execute the (one-line) statement in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a “global” command on the same line, e.g.:

```python
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

`quit()` Quit from the debugger. The program being executed is aborted.

### 9.2 How It Works

Some changes were made to the interpreter:

- `sys.settrace(func)` sets the global trace function
- there can also be a local trace function (see later)

Trace functions have three arguments: `frame, event, arg`

- `frame` is the current stack frame
- `event` is a string: 'call', 'line', 'return' or 'exception'
- `arg` is dependent on the event type

A trace function should return a new trace function or None. Class methods are accepted (and most useful!) as trace methods.

The events have the following meaning:

- **'call'**: A function is called (or some other code block entered). The global trace function is called; `arg` is the argument list to the function; the return value specifies the local trace function.
- **'line'**: The interpreter is about to execute a new line of code (sometimes multiple line events on one line exist). The local trace function is called; `arg` in None; the return value specifies the new local trace function.
- **'return'**: A function (or other code block) is about to return. The local trace function is called; `arg` is the value that will be returned. The trace function’s return value is ignored.
- **'exception'**: An exception has occurred. The local trace function is called; `arg` is a triple (exception, value, traceback); the return value specifies the new local trace function

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

Stack frame objects have the following read-only attributes:

- `f_code` the code object being executed
- `f_lineno` the current line number (-1 for 'call' events)
- `f_back` the stack frame of the caller, or None
- `f_locals` dictionary containing local name bindings
- `f_globals` dictionary containing global name bindings

Code objects have the following read-only attributes:

- `co_code` the code string
- `co_names` the list of names used by the code
- `co_consts` the list of (literal) constants used by the code
- `co_filename` the filename from which the code was compiled
Chapter 10

The Python Profiler

Copyright © 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

The profiler was written after only programming in Python for 3 weeks. As a result, it is probably clumsy code, but I don’t know for sure yet ‘cause I’m a beginner :-). I did work hard to make the code run fast, so that profiling would be a reasonable thing to do. I tried not to repeat code fragments, but I’m sure I did some stuff in really awkward ways at times. Please send suggestions for improvements to: jar@infosseek.com. I won’t promise any support ...but I’d appreciate the feedback.

10.1 Introduction to the profiler

A *profiler* is a program that describes the run time performance of a program, providing a variety of statistics. This documentation describes the profiler functionality provided in the modules *profile* and *pstats*. This profiler provides *deterministic profiling* of any Python programs. It also provides a series of report generation tools to allow users to rapidly examine the results of a profile operation.

10.2 How Is This Profiler Different From The Old Profiler?

The big changes from old profiling module are that you get more information, and you pay less CPU time. It’s not a trade-off, it’s a trade-up.

To be specific:

**Bugs removed:** Local stack frame is no longer molested, execution time is now charged to correct functions.

**Accuracy increased:** Profiler execution time is no longer charged to user’s code, calibration for platform is supported, file reads are not done by profiler during profiling (and charged to user’s code!).

**Speed increased:** Overhead CPU cost was reduced by more than a factor of two (perhaps a factor of five), lightweight profiler module is all that must be loaded, and the report generating module (*pstats*) is not needed during profiling.

**Recursive functions support:** Cumulative times in recursive functions are correctly calculated; recursive entries are counted.

**Large growth in report generating UI:** Distinct profiles runs can be added together forming a comprehensive report; functions that import statistics take arbitrary lists of files; sorting criteria is now based on keywords (instead of 4 integer options); reports shows what functions were profiled as well as what profile file was referenced; output format has been improved.

10.3 Instant Users Manual

This section is provided for users that “don’t want to read the manual.” It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile an application with a main entry point of `foo()`, you would add the following to your module:

```python
import profile
profile.run("foo()")
```

The above action would cause `foo()` to be run, and a series of informative lines (the profile) to be printed. The above approach is most useful when working with the interpreter. If you would like to save the results of a profile into a file for later examination, you can supply a file name as the second argument to the `run()` function:

```python
import profile
profile.run("foo()", 'fooprof')
```
When you wish to review the profile, you should use the methods in the \texttt{pstats} module. Typically you would load the statistics data as follows:

\begin{verbatim}
import pstats
p = pstats.Stats('fooprof')
\end{verbatim}

The class \texttt{Stats} (the above code just created an instance of this class) has a variety of methods for manipulating and printing the data that was just read into \texttt{p}. When you ran \texttt{profile.run()} above, what was printed was the result of three method calls:

\begin{verbatim}
p.strip_dirs().sort_stats(-1).print_stats()
\end{verbatim}

The first method removed the extraneous path from all the module names. The second method sorted all the entries according to the standard module/line/name string that is printed (this is to comply with the semantics of the old profiler). The third method printed out all the statistics. You might try the following sort calls:

\begin{verbatim}
p.sort_stats('name').print_stats()
\end{verbatim}

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

\begin{verbatim}
p.sort_stats('cumulative').print_stats(10)
\end{verbatim}

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

\begin{verbatim}
p.sort_stats('time').print_stats(10)
\end{verbatim}

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

\begin{verbatim}
p.sort_stats('file').print_stats('__init__')
\end{verbatim}

This will sort all the statistics by file name, and then print out statistics for only the class \texttt{init} methods ('cause they are spelled with \texttt{__init__} in them). As one final example, you could try:

\begin{verbatim}
p.sort_stats('time', 'cum').print_stats(5, 'init')
\end{verbatim}

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50\% (re: \texttt{.5}) of its original size, then only lines containing \texttt{init} are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now ('\texttt{p}' is still sorted according to the last criteria) do:

\begin{verbatim}
p.print_callers(.5, 'init')
\end{verbatim}

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

\begin{verbatim}
p.print_callees()
p.add('fooprof')
\end{verbatim}

### 10.4 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a hook (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot spots" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

### 10.5 Reference Manual

The primary entry point for the profiler is the global function \texttt{profile.run()}. It is typically used to create any profile information. The reports are formatted and printed using methods of the class \texttt{pstats.Stats}. The following is a description of all of these standard entry points and functions.

For a more in-depth view of some of the code, consider reading the later section on Profiler Extensions.
which includes discussion of how to derive “better” profilers from the classes presented, or reading
the source code for these modules.

**profile.run(string [, filename [, ...]]))**

This function takes a single argument that has can be passed to the exec statement, and an
optional file name. In all cases this routine attempts to exec its first argument, and gather
profiling statistics from the execution. If no file name is present, then this function automatically
prints a simple profiling report, sorted by the standard name (file/line/function-name) that
is presented in each line. The following is a typical output from such a call:

```python
main()
```

Ordered by: standard name

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.006</td>
<td>0.953</td>
<td>0.477</td>
<td>0.250</td>
<td>pobject.py:75(save_objects)</td>
</tr>
<tr>
<td>43/3</td>
<td>0.533</td>
<td>0.012</td>
<td>0.749</td>
<td>0.250</td>
<td>pobject.py:99(evaluate)</td>
</tr>
</tbody>
</table>

... The first line indicates that this profile was generated by the call:
```
profile.run('main()'),
```
and hence the executed string is ‘main()’. The second line indicates
that 2706 calls were monitored. Of those calls, 2004 were primitive. We define primitive to mean that
the call was not induced via recursion. The next line: Ordered by: standard name, indicates
that the text string in the far right column was used to sort the output. The column headings include:

n\text{calls} \quad \text{tottime} \quad \text{percall} \quad \text{cumtime} \quad \text{percall} \quad \text{filename:lineno(function)}

ncalls for the number of calls,
tottime for the total time spent in the given function (and excluding time made in calls to sub-functions),
percall is the quotient of tottime divided by n\text{calls}
cumtime is the total time spent in this and all subfunctions (i.e., from invocation till exit). This figure is
accurate even for recursive functions.
percall is the quotient of cumtime divided by primitive calls.
file/name/line/func name) provides the respective data of each function.

When there are two numbers in the first column (e.g., ‘43/3’), then the latter is the number of primitive
calls, and the former is the actual number of calls. Note that when the function does not recurse, these
two values are the same, and only the single figure is printed.

**pstats.stats (filename [, ...])**

This class constructor creates an instance of a “statistics object” from a filename (or set of filenames). Stats objects are manipulated by methods, in order to print useful reports.
The file selected by the above constructor must have been created by the corresponding version
of profile. To be specific, there is NO file compatibility guaranteed with future versions of this
profiler, and there is no compatibility with files produced by other profilers (e.g., the old system profiler).

If several files are provided, all the statistics for identical functions will be coalesced, so that an
overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing Stats object, the add() method can be used.

10.5.1 The Stats Class

**strip_dirs()**

This method for the Stats class removes all leading path information from file names. It is
very useful in reducing the size of the printout to fit within (close to) 80 columns. This method
modifies the object, and the stripped information is lost. After performing a strip operation, the
object is considered to have its entries in a “random” order, as it was just after object initialization
and loading. If strip_dirs() causes two function names to be indistinguishable (i.e., they
are on the same line of the same filename, and have the same function name), then the statistics
for these two entries are accumulated into a single entry.

**add (filename [, ...])**

This method of the Stats class accumulates additional profiling information into the current
profiling object. Its arguments should refer to filenames created by the corresponding version
of profile.run(). Statistics for identically named (i.e., file, line, name) functions are
automatically accumulated into single function statistics. Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (i.e., alphabetical).

The subtle distinction between *nfl* and *stdname* is that the standard name is a sort of
the name as printed, which means that the embedded line numbers get compared in an odd way.
For example, lines 3, 20, and 40 would (if the filenames were the same) appear in the string
order 20, 3 and 40. In contrast, *sort_stats("nfl")* does a numeric compare of the line numbers. In fact,
*sort_stats("nfl")* is the same as sort_stats("name", "file", "line").

For compatibility with the old profiler, the numeric arguments ‘-1’, ‘0’, ‘1’, and ‘2’ are permitted. They are interpreted as "stdname", "calls", "time", and "cumulative".
respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()
This method for the Stats class reverses the ordering of the basic list within the object. This method is provided primarily for compatibility with the old profiler. Its utility is questionable now that ascending vs descending order is properly selected based on the sort key of choice.

print_stats(restrictions [...])
This method for the Stats class prints out a report as described in the profile.run() definition.

The order of the printing is based on the last sort_stats() operation done on the object (subject to caveats in add() and strip_dirs()).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

\[
\text{print_stats}(1, \text{"foo:"})
\]
would first limit the printing to first 10% of list, and then only print functions that were part of a filename `foo:`. In contrast, the command:

\[
\text{print_stats}(\text{"foo:"}, 0.1)
\]
would limit the list to all functions having file names `foo:` and then proceed to only print the first 10% of them.

print_callers(restrictions [...])
This method for the Stats class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by print_stats(), and the definition of the restricting argument is also identical. For convenience, a number is shown in parentheses after each caller to show how many times this specific call was made. A second non-parenthesized number is the cumulative time spent in the function at the right.

print_callees(restrictions [...])
This method for the Stats class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the print_callers() method.

Ignore()
This method of the Stats class is used to dispose of the value returned by earlier methods. All standard methods in this class return the instance that is being processed, so that the commands can be strung together. For example:

\[
\text{print_stats('foo file').strip_dirs().sort_stats('cum')} \\n\text{.print_stats().ignore()}
\]
would perform all the indicated functions, but it would not return the final reference to the Stats instance.

10.6 Limitations

There are two fundamental limitations on this profiler. The first is that it relies on the Python interpreter to dispatch call, return, and exception events. Compiled C code does not get interpreted, and hence is "invisible" to the profiler. All time spent in C code (including built-in functions) will be charged to the Python function that invoked the C code. If the C code calls out to some native Python code, then those calls will be profiled properly.

The second limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about 0.001 seconds. Hence no measurements will be more accurate that that underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error...

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (i.e., less than one clock tick), but it can accumulate and become very significant. This profiler provides a means of calibrating itself for a given platform so that this error can be probabilistically (i.e., on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-). Do NOT be alarmed by negative numbers in the profile. They should only appear if you have calibrated your profiler, and the results are actually better than without calibration.

10.7 Calibration

The profiler class has a hard coded constant that is added to each event handling time to compensate for the overhead of calling the time function, and socking away the results. The following procedure can be used to obtain this constant for a given platform (see discussion in section Limitations above).

\[
\text{import profile} \\
\text{pr = profile.Profile()} \\
\text{pr.calibrate(100)} \\
\text{pr.calibrate(100)} \\
\text{pr.calibrate(100)}
\]

The argument to calibrate() is the number of times to try to do the sample calls to get the CPU times. If your computer is very fast, you might have to do:

\[
\text{pr.calibrate(1000)}
\]
or even:
pr_calibrate(10000)

The object of this exercise is to get a fairly consistent result. When you have a consistent answer, you are ready to use that number in the source code. For a Sun Sparcstation 1000 running Solaris 2.3, the magical number is about .00053. If you have a choice, you are better off with a smaller constant, and your results will “less often” show up as negative in profile statistics.

The following shows how the trace_dispatch() method in the Profile class should be modified to install the calibration constant on a Sun Sparcstation 1000:

```python
def trace_dispatch(self, frame, event, arg):
    t = self.timer()
    t = t[0] + t[1] - .00053 # Calibration constant
    if self.dispatch(event, frame, t):
        t = self.timer()
        self.t = t[0] + t[1]
    else:
        r = self.timer()
        self.t = r[0] + r[1] - t # put back unrecorded delta
    return
```

Note that if there is no calibration constant, then the line containing the calibration constant should simply say:

```python
t = t[0] + t[1] - self.t # no calibration constant
```

You can also achieve the same results using a derived class (and the profiler will actually run equally fast!), but the above method is the simplest to use. I could have made the profiler “self calibrating”, but it would have made the initialization of the profiler class slower, and would have required some very fancy coding, or else the use of a variable where the constant ‘.00053’ was placed in the code shown. This is a VERY critical performance section, and there is no reason to use a variable lookup at this point, when a constant can be used.

10.8 Extensions — Deriving Better Proﬁlers

The Profile class of module profile was written so that derived classes could be developed to extend the profiler. Rather than describing all the details of such an effort, I’ll just present the following two examples of derived classes that can be used to do proﬁling. If the reader is an avid Python programmer, then it should be possible to use these as a model and create similar (and perhaps better) proﬁle classes.

If all you want to do is change how the timer is called, or which timer function is used, then the basic class has an option for that in the constructor for the class. Consider passing the name of a function to call into the constructor:

```python
pr = profile.Profile(your_time_func)
```

The resulting proﬁler will call your_time_func() instead of os.times(). The function should return either a single number or a list of numbers (like what os.times() returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the proﬁler class for the timer function that you choose. For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during proﬁling. (os.times is pretty bad, ‘cause it returns a tuple of ﬂoating point values, so all arithmetic is ﬂoating point in the proﬁler!). If you want to substitute a better timer in the cleanest fashion, you should derive a class, and simply put in the replacement dispatch method that better handles your timer call, along with the appropriate calibration constant.

10.8.1 OldProfile Class

The following derived proﬁler simulates the old style proﬁler, providing errant results on recursive functions. The reason for the usefulness of this proﬁler is that it runs faster (i.e., less overhead) than the old proﬁler. It still creates all the caller stats, and is quite useful when there is no recursion in the user’s code. It is also a lot more accurate than the old proﬁler, as it does not change all its overhead time to the user’s code.

```python
class OldProfile(Profile):
    def traceDispatch_except(self, frame, t):
        rt, rtt, rct, rfn, rframe, rcur = self.cur
        if rcur and not rframe is frame:
            return self.traceDispatch_return(rframe, t)
        return 0

    def traceDispatch_call(self, frame, t):
        fn = ‘frame.f_code’
        self.cur = (t, 0, 0, fn, frame, self.cur)
        if self.timings.has_key(fn):
            tt, ct, callers = self.timings[fn]
            self.timings[fn] = tt, ct, callers
        else:
            self.timings[fn] = 0, 0, {}
        return 1

    def traceDispatch_return(self, frame, t):
        rt, rtt, rct, rfn, frame, rcur = self.cur
        rtt = rtt + t
        sft = rtt + rct
```
pt, ptt, pct, pfn, pframe, pcur = rcur
self.cur = pt, ptt+rt, pct+sft, pfn, pframe, pcur

tt, ct, callers = self.timings[rfn]
if callers.has_key(pfn):
    callers[pfn] = callers[pfn] + 1
else:
    callers[pfn] = 1
self.timings[rfn] = tt+rtt, ct + sft, callers

return 1

def snapshot_stats(self):
    self.stats = {}
    for func in self.timings.keys():
        tt, ct, callers = self.timings[func]
        nor_func = self.func_normalize(func)
        nor_callers = {}
        nc = 0
        for func_caller in callers.keys():
            nor_callers[self.func_normalize(func_caller)] = \
                callers[func_caller]
        nc = nc + nor_callers[nor_caller]
        self.stats[nor_func] = nc, nc, tt, ct, nor_callers

10.8.2 HotProfile Class

This profiler is the fastest derived profile example. It does not calculate caller-callee relationships, and does not calculate cumulative time under a function. It only calculates time spent in a function, so it runs very quickly (rc: very low overhead). In truth, the basic profiler is so fast, that is probably not worth the savings to give up the data, but this class still provides a nice example.

class HotProfile(Profile):

def trace_dispatch_exception(self, frame, t):
    rt, rtt, rfn, rframe, rcur = self.cur
    if rcur and not rframe is frame:
        return self.trace_dispatch_return(rframe, t)
    return 0

def trace_dispatch_call(self, frame, t):
    self.cur = (t, 0, frame, self.cur)
    return 1

def trace_dispatch_return(self, frame, t):
    rt, rtt, frame, rcur = self.cur
    rfn = 'frame.f_code'
    pt, ptt, pframe, pcur = rcur
    self.cur = pt, ptt+rt, pframe, pcur

if self.timings.has_key(rfn):
    nc, tt = self.timings[rfn]
    self.timings[rfn] = nc + 1, rt + rtt + tt
else:
    self.timings[rfn] = 1, rt + rtt

return 1
Chapter 11

Internet and WWW Services

The modules described in this chapter provide support for a variety of subprotocols used in the World-Wide-Web (WWW) environment. Some of these modules require the presence (either as a built-in module or as a separate package) of the socket module, which is currently only fully supported on Unix and Windows NT. Here is an overview:

- **cgi** — Common Gateway Interface, used to interpret forms in server-side scripts.
- **urllib** — Open an arbitrary object given by URL (requires sockets).
- **httplib** — HTTP protocol client (requires sockets).
- **ftplib** — FTP protocol client (requires sockets).
- **gopherlib** — Gopher protocol client (requires sockets).
- **nntplib** — NNTP protocol client (requires sockets).
- **urlparse** — Parse a URL string into a tuple (addressing scheme identifier, network location, path, parameters, query string, fragment identifier).
- **htmllib** — A (slow) parser for HTML files.
- **sgmllib** — Only as much of an SGML parser as needed to parse HTML.
- **rfc822** — Parse RFC-822 style mail headers.
- **mimetools** — Tools for parsing MIME style message bodies.

11.1 Standard Module **cgi**

This module makes it easy to write Python scripts that run in a WWW server using the Common Gateway Interface. It is used by Michael McGay and subsequently modified by Steve Majeck.

When a WWW server finds that a URL contains a reference to a file in a particular subdirectory (usually /cgi-bin or /serve as a subdirectory), it reads the file as a subprocess. Information about the request (such as the full URL, the client's host, etc.) is passed in the environment of the subprocess. The output from the subprocess is sent to the client across the network, to the client as the response from the WWW server. The output from the subprocess is captured in the environment variables passed to the script. The scripting language for the CGI protocol is described below.

The CGI protocol defines several important concepts that make it easy to interpret the information passed in the environment variables passed to the subprocesses from a WWW server. In particular, the CGI protocol is a set of standards for submitting forms in the HTTP protocol.

The formatting of the output is so trivial that it is almost always used. All you need to do is print a minimal set of MIME headers describing the output format, followed by a blank line and your actual output. If you want to generate HTML, your script could start as follows:

```python
# Header -- one or more lines:
print "Content-type: text/html"
# Blank line separating header from body:
print
# Body, in HTML format:
print "<TITLE>" The Amazing SPAM Homepage! "</TITLE> "
# etc...
```

The server will add some header lines of its own, but it won't touch the output following the header. The CGI module defines the following functions:

- `parse()` — Read and parse the form submitted to the script and return a dictionary containing the form's fields. This function is called at most once per script invocation, as it may consume standard input (if the form was submitted through a POST request). The dictionary keys are the names used in the submission; the values are the values of the fields. The function will also set the `QUERY_STRING` environment variable if it isn't already set.
- `print_form(form)` — Print a piece of HTML showing the contents of the form (a dictionary).
- `print_environ()` — Print a piece of HTML showing the entire contents of the server environment. This is mainly useful when learning about writing CGI scripts.
- `print_var_name(var)` — Print a piece of HTML using the environment variables that may be set by the CGI protocol.

The module is described in the chapter and is available in World-Wide-Web (WWW) clients and servers, and a few modules related to news and email. They are all implemented in Python, which is currently only fully supported on Unix and Windows NT. Here is an overview:
FormContentDict class defined below, or a subclass thereof). This is mainly useful when debugging a CGI script.

\texttt{escape(string)}

Convert special characters in \texttt{string} to HTML escapes. In particular, "&" is replaced with "\&", "<" is replaced with "\lt;", and ">" is replaced with "\gt;". This is useful when printing (almost) arbitrary text in an HTML context. Note that for inclusion in quoted tag attributes (e.g. \texttt{<A HREF="...">}), some additional characters would have to be converted — in particular the string quote. There is currently no function that does this.

The module defines the following classes. Since the base class initializes itself by calling \texttt{parse()}, at most one instance of at most one of these classes should be created per script invocation:

\texttt{FormContentDict()}

This class behaves like a (read-only) dictionary and has the same keys and values as the dictionary returned by \texttt{parse()} (i.e. each field name maps to a list of values). Additionally, it initializes its data member \texttt{query_string} to the raw query sent from the server.

\texttt{SvFormContentDict()}

This class, derived from \texttt{FormContentDict}, is a little more user-friendly when you are expecting that each field name is only used once in the form. When you access for a particular field (using \texttt{form[fieldname]}), it will return the string value of that item if it is unique, or raise \texttt{IndexError} if the field was specified more than once in the form. (If the field wasn't specified at all, \texttt{KeyError} is raised.) To access fields that are specified multiple times, use \texttt{form.getlist(fieldname)}. The \texttt{values()} and \texttt{items()} methods return mixed lists — containing strings for singly-defined fields, and lists of strings for multiply-defined fields.

(It currently defines some more classes, but these are experimental and/or obsolescent, and are thus not documented — see the source for more informations.)

The module defines the following variable:

\texttt{environ}

The shell environment, exactly as received from the http server. See the CGI documentation for a description of the various fields.

11.1.1 Example

This example assumes that you have a WWW server up and running, e.g. NCSA's \texttt{httpd}.

Place the following file in a convenient spot in the WWW server's directory tree. E.g., if you place it in the subdirectory 'test' of the root directory and call it 'test.html', its URL will be 'http://yourservename/test/test.html'.

\begin{verbatim}
<TITLE>Test Form Input</TITLE>
<FORM METHOD="POST" ACTION="/cgi-bin/test.py"
<INPUT NAME="Name" (Name)<br>
<INPUT TYPE=SUBMIT>
</FORM>
\end{verbatim}

Selecting this file's URL from a forms-capable browser such as Mosaic or Netscape will bring up a simple form with two text input fields and a "submit" button.

But wait. Before pressing "submit", a script that responds to the form must also be installed. The test file as shown assumes that the script is called 'test.py' and lives in the server's \texttt{cgi-bin} directory. Here's the test script:

\begin{verbatim}
#!/usr/local/bin/python
import cgi
print "Content-type: text/html"
print # End of headers!
print "<TITLE>Test Form Output</TITLE>"
print "<H1>Test Form Output</H1>"
form = cgi.SvFormContentDict() # Load the form
name = addr = None # Default: no name and address
# Extract name and address from the form, if given
if form.has_key('Name'):
    name = form['Name']
if form.has_key('Address'):
    addr = form['Address']
# Print an unnumbered list of the name and address, if present
print "<UL>
if name is not None:
    print "<LI>Name:", cgi.escape(name)
if addr is not None:
    print "<LI>Address:", cgi.escape(addr)
print "</UL>"
\end{verbatim}

The script should be made executable ('\texttt{chmod +x script}') If the Python interpreter is not located at '/usr/local/bin/python' but somewhere else, the first line of the script should be modified accordingly.

Now that everything is installed correctly, we can try out the form. Bring up the test form in your WWW browser, fill in a name and address in the form, and press the "submit" button. The script should now run and its output is sent back to your browser. This should roughly look as follows:

Test Form Output

- **Name**: the name you entered
- **Address**: the address you entered
If you didn’t enter a name or address, the corresponding line will be missing (since the browser doesn’t send empty form fields to the server).

11.2 Standard Module urllib

This module provides a high-level interface for fetching data across the World-Wide Web. In particular, the urlopen function is similar to the built-in function open, but accepts URLs (Universal Resource Locators) instead of filenames. Some restrictions apply — it can only open URLs for reading, and no seek operations are available.

It defines the following public functions:

urlopen (url)

Open a network object denoted by a URL for reading. If the URL does not have a scheme identifier, or if it has ‘file:’ as its scheme identifier, this opens a local file; otherwise it opens a socket to a server somewhere on the network. If the connection cannot be made, or if the server returns an error code, the IOError exception is raised. If all went well, a file-like object is returned. This supports the following methods: read(), readline(), readlines(), fileno(), close() and info(). Except for the last one, these methods have the same interface as for file objects — see the section on File Objects earlier in this manual. (It’s not a built-in file object, however, so it can’t be used at those few places where a true built-in file object is required.)

The info() method returns an instance of the class rfc822.Message containing the headers received from the server; if the protocol uses such headers (currently the only supported protocol that uses this is HTTP). See the description of the rfc822 module.

urllib2.urlopen()

Copy a network object denoted by a URL to a local file, if necessary. If the URL points to a local file, or a valid cached copy of the object exists, the object is not copied. Return a tuple (filename, headers) where filename is the local file name under which the object can be found, and headers is either None (for a local object) or whatever the info() method of the object returned by urlopen() returned (for a remote object, possibly cached). Exceptions are the same as for urlopen().

urlcleanup ()

Clear the cache that may have been built up by previous calls to urlopen().

quote (string [, addsafe ])

Replace special characters in string using the %xx escape. Letters, digits, and the characters ",", "", "" are never quoted. The optional addsafe parameter specifies additional characters that should not be quoted — its default value is '/'.

Example: quote('"connolly/') yields '%7Econnolly/'

unquote (string)

Replace %xx escapes by their single-character equivalent.

Example: unquote('%7Econnolly/') yields '"connolly/'

Restrictions:

- Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), Gopher (but not Gopher+), FTP, and local files.
- The caching feature of urllib2.urlopen() has been disabled until I find the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can’t be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The urlopen() and urllist() functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.
- The data returned by urlopen() or urllist() is the raw data returned by the server. This may be binary data (e.g. an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the Content-type header. For the Gopher protocol, type information is encoded in the URL; there is currently no easy way to extract it. If the returned data is HTML, you can use the module htmllib to parse it.
- Although the urllib module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module urllib2.

11.3 Standard Module httplib

This module defines a class which implements the client side of the HTTP protocol. It is normally not used directly — the module urllib uses it to handle URLs that use HTTP.

The module defines one class, HTTP. An HTTP instance represents one transaction with an HTTP server. It should be instantiated passing a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form host:port, else the default HTTP port (80) is used. If no host is passed, no connection is made, and the connect method should be used to connect to a server. For example, the following all create instances that connect to the server at the same host and port:

```python
>>> h1 = httplib.HTTP('www.cwi.nl')
>>> h2 = httplib.HTTP('www.cwi.nl:80')
>>> h3 = httplib.HTTP('www.cwi.nl', 80)
```

Once an HTTP instance has been connected to an HTTP server, it should be used as follows:

1. Make exactly one call to the putrequest() method.
2. Make zero or more calls to the putheader() method.
3. Call the endheaders() method (this can be omitted if step 4 makes no calls).
4. Optional calls to the send() method.
5. Call the getreply() method.
6. Call the getfile() method and read the data off the file object that it returns.

11.3.1 HTTP Objects

HTTP instances have the following methods:

- set_debuglevel(level)
  
  Set the debugging level (the amount of debugging output printed). The default debug level is 0, meaning no debugging output is printed.

- connect(host[, port])
  
  Connect to the server given by host and port. See the intro for the default port. This should be called directly only if the instance was instantiated without passing a host.

- send(data)
  
  Send data to the server. This should be used directly only after the endheaders() method has been called and before getreply() has been called.

- putrequest(request, selector)
  
  This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the request string, the selector string, and the HTTP version (HTTP/1.0).

- putheader(header, argument[, ...])
  
  Send an RFC-822 style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

- endheaders()
  
  Send a blank line to the server, signalling the end of the headers.

- getreply()
  
  Complete the request by shutting down the sending end of the socket, read the reply from the server, and return a tuple (replycode, message, headers). Here replycode is the integer reply code from the request (e.g., 200 if the request was handled properly); message is the message string corresponding to the reply code; and header is an instance of the class rfc822.Message containing the headers received from the server. See the description of the rfc822 module.

- getfile()
  
  Return a file object from which the data returned by the server can be read, using the read(), readline() or readlines() methods.

11.3.2 Example

Here is an example session:

```python
>>> import httplib
>>> h = httplib.HTTP('www.cwi.nl')
>>> h.putrequest('GET', '/index.html')
>>> h.putheader('Accept', 'text/html')
>>> h.putheader('Accept', 'text/plain')
>>> h.endheaders()
>>> errcode, errmsg, headers = h.getreply()
>>> print errcode # Should be 200
>>> f = h.getfile()
>>> data = f.read() # Get the raw HTML
>>> f.close()
```

11.4 Standard Module ftplib

This module defines the class FTP and a few related items. The FTP class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other ftp servers. It is also used by the module urllib to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet RFC 959.

Here's a sample session using the ftplib module:

```python
>>> from ftplib import FTP
>>> ftp = FTP('ftp.cwi.nl') # connect to host, default port
>>> ftp.login() # user anonymous, passwd user@hostname
>>> ftp.retrlines('LIST') # list directory contents
```

The module defines the following items:

```python
FTP([host[, user, passwd, acct]])
```

- Return a new instance of the FTP class. When host is given, the method call connect(host) is made. When user is given, additionally the method call login(user, passwd, acct) is made (where passwd and acct default to the empty string when not given).

all_errors

The set of all exceptions (as a tuple) that methods of FTP instances may raise as a result of problems with the FTP connection (as opposed to program-ending errors made by the caller). This set includes the four exceptions listed below as well as socket.error and IOError.

```python
>>> errcode, errmsg, headers = h.getreply()
>>> print errcode # Should be 200
>>> f = h.getfile()
>>> data = f.read() # Get the raw HTML
>>> f.close()
```
Error_reply
Exception raised when an unexpected reply is received from the server.

Error_temp
Exception raised when an error code in the range 400–499 is received.

Error_perm
Exception raised when an error code in the range 500–599 is received.

Error_proto
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

11.4.1 FTP Objects

FTP instances have the following methods:

set_debuglevel(level)
Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

connect(host[, port])
Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

getwelcome()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

login([user[, passwd[, acct]]])
Log in as the given user. The passwd and acct parameters are optional and default to the empty string. If no user is specified, it defaults to “Anonymous”. If user is anonymous, the default passwd is “realuser@host” where realuser is the real user name (glanced from the ‘LOGNAME’ or ‘USER’ environment variable) and host is the hostname as returned by socket.gethostname(). This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in.

abort()
Abort a file transfer that is in progress. Using this does not always work, but it’s worth a try.

sendcmd(command)
Send a simple command string to the server and return the response string.

voidcmd(command)
Send a simple command string to the server and handle the response. Return nothing if a response code in the range 200–299 is received. Raise an exception otherwise.

retrbinary(command, callback, maxblocksize)
Retrieve a file in binary transfer mode. command should be an appropriate ‘RETR’ command, i.e. “RETR filename”. The callback function is called for each block of data received, with a single string argument giving the data block. The maxblocksize argument specifies the maximum block size (which may not be the actual size of the data blocks passed to callback).

retrlines(command[, callback])
Retrieve a file or directory listing in ASCII transfer mode. varcommand should be an appropriate ‘RETR’ command (see retrbinary()) or a ‘LIST’ command (usually just the string “LIST”). The callback function is called for each line, with the trailing CRLF stripped. The default callback prints the line to sys.stdout.

storbinary(command, file, blocksize)
Store a file in binary transfer mode. command should be an appropriate ‘STOR’ command, i.e. “STOR filename”. file is an open file object which is read until EOF using its readline() method in blocks of size blocksize to provide the data to be stored.

storlines(command, file)
Store a file in ASCII transfer mode. command should be an appropriate ‘STOR’ command (see storbinary()). Lines are read until EOF from the open file object file using its readline() method to provide the data to be stored.

nlst([argument[, ...]])
Return a list of files as returned by the ‘NLST’ command. The optional argument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the ‘NLST’ command.

dir([argument[, ...]])
Return a directory listing as returned by the ‘LIST’ command, as a list of lines. The optional argument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the ‘LIST’ command. If the last argument is a function, it is used as a callback function as for retrlines().

rename(fromname, toname)
Rename file fromname on the server to toname.

cwd(pathname)
Set the current directory on the server.

mkdir(pathname)
Create a new directory on the server.

pwd()
Return the pathname of the current directory on the server.

quit()
Send a ‘QUIT’ command to the server and close the connection. This is the “polite” way to close a connection, but it may raise an exception of the server responds with an error to the QUIT command.
11.5 Standard Module `gopherlib`

This module provides a minimal implementation of client side of the the Gopher protocol. It is used by the module `urllib` to handle URLs that use the Gopher protocol.

The module defines the following functions:

```python
send_selector(selector, host [, port])
```

Send a selector string to the gopher server at host and port (default 70). Return an open file object from which the returned document can be read.

```python
send_query(selector, query, host [, port])
```

Send a selector string and a query string to a gopher server at host and port (default 70). Return an open file object from which the returned document can be read.

Note that the data returned by the Gopher server can be of any type, depending on the first character of the selector string. If the data is text (first character of the selector is '0'), lines are terminated by CRLF, and the data is terminated by a line consisting of a single '.', and a leading '.', should be stripped from lines that begin with '. '. Directory listings (first character of the selector is '1') are transferred using the same protocol.

11.6 Standard Module `nntplib`

This module defines the class `NNTP` which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```python
>>> s = NNTP('news.cwi.nl')
>>> resp, count, first, last, name = s.group('comp.lang.python')
Group comp.lang.python has 59 articles, range 3742 to 3803
>>> for id, sub in subs[-10:]: print id, sub
```

11.6.1 NNTP Objects

NNTP instances have the following methods. The `getwelcome` method returns the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

```python
getwelcome()  # Return the welcome message sent by the server in reply to the initial connection.
```

Set the instance’s debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

```python
set_debuglevel([level])  # Set the instance’s debugging level.
```

Send a ‘NEWGROUPS’ command. The date argument should be a string of the form “ymd” indicating the date, and time should be a string of the form “hmmss” indicating the time. Return a pair (response, groups) where groups is a list of group names that are new since the given date and time.

```python
newgroups(date [, time])  # Send a ‘NEWGROUPS’ command.
```
newnews(group, date, time)  
Send a 'NEWNEWS' command. Here, group is a group name or "*", and date and time have the same meaning as for newgroups(). Return a pair (response, articles) where articles is a list of article ids.

list()  
Send a 'LIST' command. Return a pair (response, list) where list is a list of tuples. Each tuple has the form (group, last, first, stat, flag), where group is a group name, last and first are the last and first article numbers (as strings), and flag is 'y' if posting is allowed, 'n' if not, and 'k' if the newsgroup is moderated. (Note the ordering last, first.)

group(name)  
Send a 'GROUP' command, where name is the group name. Return a tuple (response, count, first, last, name) where count is the (estimated) number of articles in the group, first is the first article number in the group, last is the last article number in the group, and name is the group name. The numbers are returned as strings.

help()  
Send a 'HELP' command. Return a pair (response, list) where list is a list of help strings.

stat(id)  
Send a 'STAT' command, where id is the message id (enclosed in '<' and '>') or an article number (as a string). Return a tuple (response, number, id) where number is the article number (as a string) and id is the article id (enclosed in '<' and '>').

next()  
Send a 'NEXT' command. Return as for stat().

last()  
Send a 'LAST' command. Return as for stat().

head(id)  
Send a 'HEAD' command, where id has the same meaning as for stat(). Return a pair (response, list) where list is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

body(id)  
Send a 'BODY' command, where id has the same meaning as for stat(). Return a pair (response, list) where list is a list of the article's body text (an uninterpreted list of lines, without trailing newlines).

article(id)  
Send a 'ARTICLE' command, where id has the same meaning as for stat(). Return a pair (response, list) where list is a list of the article's body text (an uninterpreted list of lines, without trailing newlines).

slave()  
Send a 'SLAVE' command. Return the server's response.

xhdr(header, string)  
Send an 'XHDR' command. This command is not defined in the RFC but is a common extension. The header argument is a header keyword, e.g. "subject". The string argument should have the form "first-last" where first and last are the first and last article numbers to search. Return a pair (response, list), where list is a list of pairs (id, text), where id is an article id (as a string) and text is the text of the requested header for that article.

post(file)  
Post an article using the 'POST' command. The file argument is an open file object which is read until EOF using its readline() method. It should be a well-formed news article, including the required headers. The post() method automatically escapes lines beginning with '\'.

ihave(id, file)  
Send an 'IHAVE' command. If the response is not an error, treat file exactly as for the post() method.

quit()  
Send a 'QUIT' command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

11.7 Standard Module urlparse

This module defines a standard interface to break URL strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL".

The module has been designed to match the current Internet draft on Relative Uniform Resource Locators (and discovered a bug in an earlier draft!).

It defines the following functions:

urlparse(urlstring[, default_scheme[, allow_fragments]])  
Parse a URL into 6 components, returning a 6-tuple: (addressing scheme, network location, path, parameters, query, fragment identifer). This corresponds to the general structure of a URL: "scheme://netloc/path?parameters#query?fragment". Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (e.g. the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the tuple items, except for a leading slash in the path component, which is retained if present.

Example:

urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
yields the tuple

('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '', '', '').

If the default_scheme argument is specified, it gives the default addressing scheme, to be used only if the URL string does not specify one. The default value for this argument is the empty string.

If the allow_fragments argument is zero, fragment identifiers are not allowed, even if the URL's addressing scheme normally does support them. The default value for this argument is 1.

urlunparse(tupletuple)  
Construct a URL string from a tuple as returned by urlparse. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had redundant delimiters, e.g. a ? with an empty query (the draft states that these are equivalent).

urljoin(base, url[, allow_fragments])  
Construct a full ("absolute") URL by combining a "base URL" (base) with a "relative URL" (url). Informally, this uses components of the base URL in particular the addressing scheme, the network location (and part of) the path, to provide missing components in the relative URL.

Example:

urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
yields the string

'http://www.cwi.nl/%7Eguido/FAQ.html'.

The allow_fragments argument has the same meaning as for urlparse.
### 11.8 Standard Module `htmlib`

This module defines a number of classes which can serve as a basis for parsing text files formatted in HTML (HyperText Markup Language). The classes are not directly concerned with I/O — the have to be fed their input in string form, and will make calls to methods of a “formatter” object in order to produce output. The classes are designed to be used as base classes for other classes in order to add functionality, and allow most of their methods to be extended or overridden. In turn, the classes are derived from and extend the class `SGMLParser` defined in module `sgmllib`.

The following is a summary of the interface defined by `sgmllib.SGMLParser`:

- **The interface to feed data to an instance is through the `feed()` method, which takes a string argument.**
  - This can be called with as little or as much text at a time as desired; `p.feed(a+b)` has the same effect as `p.feed(a)+p.feed(b)`. When the data contains complete HTML elements, these are processed immediately; incomplete elements are saved in a buffer. To force processing of all unprocessed data, call the `close()` method.
  - Example: to parse the entire contents of a file, do `p.feed(open(file).read())`; `p.close()`.

- **The interface to define semantics for HTML tags is very simple:** derive a class and define methods called `start_tag()`, `end_tag()`, or `do_tag()`. The parser will call these at appropriate moments: `start_tag` or `do_tag` is called when an opening tag of the form `<tag> ... </tag>` is encountered; `end_tag` is called when a closing tag of the form `<tag>` is encountered. If an opening tag requires a corresponding closing tag, like `<H1> ... </H1>`, the class should define the `start_tag` method; if a tag requires no closing tag, like `<P>`, the class should define the `do_tag` method.

The module defines the following classes:

#### `HTMLParser`

This class is the most basic HTML parser class. It defines a single `entity` name over the names defined by the `SGMLParser` base class, `&bullet;`. It also defines handles for the following tags: `<LISTING>`, `</LISTING>`, `<XMP>`, `</XMP>`, and `<PLAINTEXT>` (the latter is terminated only by end of file).

#### `CollectingParser`

This class, derived from `HTMLParser`, collects various useful bits of information from the HTML text. To this end it defines additional handlers for the following tags: `<a>`, ... </a>, `<head>`, ... </head>, `<body>`, ... </body>, `<title>`, ... </title>, and `<isindex>`.

#### `FormattingParser` (**format**, **stylesheet**)

This class, derived from `CollectingParser`, interprets a wide selection of HTML tags so it can produce formatted output from the parsed data. It is initialized with two objects, a `formatter` which should define a number of methods to format text into paragraphs, and a `stylesheet` which defines a number of style parameters for the formatting process. Formatters and style sheets are documented later in this section.

#### `AnchoringParser` (**format**, **stylesheet**)

This class, derived from `FormattingParser`, extends the handling of the `<a>` ... </a> tag pair to call the formatter’s `bgn_anchor()` and `end_anchor()` methods. This allows the formatter to display the anchor in a different font or color, etc.

Instances of `CollectingParser` (and thus also instances of `FormattingParser` and `AnchoringParser`) have the following instance variables:

- **anchors**
  - A list of the values of the `NAME` attributes of the `<a>` tags encountered.

- **anchortypes**
  - A list of the values of the `TYPE` attributes of the `<a>` tags encountered.

- **anchornames**
  - A list of the values of `HREF` attributes of the `<a>` tags encountered.

- **inanchor**
  - Outside an `<a>` ... </a> tag pair, this is zero. Inside such a pair, it is a unique integer, which is positive if the anchor has a `HREF` attribute, negative if it hasn’t. Its absolute value is one more than the index of the anchor in the anchors, anchornames and anchortypes lists.

- **isindex**
  - True if the `<ISINDEX>` tag has been encountered.

- **nextid**
  - The attribute list of the last `<NEXTID>` tag encountered, or an empty list if none.

- **title**
  - The text inside the last `<TITLE>` ... </TITLE> tag pair, or ‘‘’ if no title has been encountered yet.

The module also defines a number of style sheet classes. These should never be instantiated — their class variables are the only behavior required. Note that style sheets are specifically designed for a particular formatter implementation. The currently defined style sheets are:

- **NullStylesheet**
  - A style sheet for use on a dumb output device such as an ASCII terminal.

- **X11Stylesheet**
  - A style sheet for use with an X11 server.

- **MacStylesheet**
  - A style sheet for use on Apple Macintosh computers.

- **StdwinStylesheet**
  - A style sheet for use with the stdwin module; it is an alias for either X11Stylesheet or MacStylesheet.

- **GLStylesheet**
  - A style sheet for use with the SGI Graphics Library and its font manager (the SGI-specific built-in modules gl and fm).

Style sheets have the following class variables:

- **stdfontset**
  - A list of up to four font definitions, respectively for the roman, italic, bold and constant-width variant of a font for normal text. If the list contains less than four font definitions, the last item is used as the default for missing items. The type of a font definition depends on the formatter in use; it’s only use is as a parameter to the formatter’s `setfont()` method.

- **hifontset**
  - **h2fontset**
  - **h3fontset**
  - **h4fontset**
  - The font set used for various headers (text inside `<H1>`, ... </H1>) tag pairs etc.

- **stdindent**
  - The indentation of normal text. This is measured in the “native” units of the formatter in use; for some formatters these are characters, for others (especially those that actually support variable-spacing fonts)
in pixels or printer points.

- ddindent
  - The indentation used for the first level of \('<DD>' tags.

- ulindent
  - The indentation used for the first level of \('<UL>' tags.

- h1indent
  - The indentation used for level 1 headers.

- h2indent
  - The indentation used for level 2 headers.

- literalindent
  - The indentation used for literal text (text inside \('<PLAINTEXT>' and similar tag pairs).

- character references of the form \('&#name;').

- SGML comments of the form \('<!-- text -->').

- The indentation used for the first level of \('<DD>' tags.

- Set the font to be used subsequently. The \(\text{fontspec}\) argument is an item in a style sheet’s font set.

- Finish the current line, if not empty, and begin a new one.

- Set the left indentation of the following lines to \(n\) units.

- Require at least \(n\) blank lines before the next line. Implies flush().

- Add a word to the current paragraph, followed by \(n\) spaces.

- If this instance variable is true, empty words should be ignored by addword. It should be set to false after a non-empty word has been added.

- Set the justification of the current paragraph. The justification can be 'c' (center), 'l' (left justified), 'r' (right justified) or 'lr' (left and right justified).

- Begin an anchor. The \(id\) parameter is the value of the parser’s \(\text{inanchor}\) attribute.

- End an anchor. The \(id\) parameter is the value of the parser’s \(\text{inanchor}\) attribute.

A sample formatter implementation can be found in the module \fmt\, which in turn uses the module \Para\.

These modules are not intended as standard library modules; they are available as an example of how to write a formatter.

11.9 Standard Module sgmllib

This module defines a class \SGMLParser\ which serves as the basis for parsing text files formatted in SGML (Standard Generalized Mark-up Language). In fact, it does not provide a full SGML parser — it only parses SGML insofar as it is used by HTML, and the module only exists as a basis for the \htmlib\ module.

In particular, the parser is hardcoded to recognize the following elements:

- Opening and closing tags of the form \('<tag attr="value" ...>') and \('<\tag>'\), respectively.
- Character references of the form \('&\#name;'>.
- Entity references of the form \('&name;'>.
- SGML comments of the form \('<!--new-->').

The \SGMLParser\ class must be instantiated without arguments. It has the following interface methods:

- \(\text{reset}()\)
  - Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.
- \(\text{setnonemptytags}()\)
  - Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML tag \('<PLAINTEXT>' can be implemented.)

- \(\text{fe re}()\)
  - Feed some text to the parser. It is processed as if it consists of complete elements; incomplete data is buffered until more data is fed or \(\text{close}()\) is called.

- \(\text{close}()\)
  - Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call \(\text{SGMLParser.close}()\).

- \(\text{handle_charref}(\text{ref})\)
  - This method is called to process a character reference of the form \('&\#ref;'> where \(\text{ref}\) is a decimal number in the range 0-255. It translates the character to ASCII and calls the method \(\text{handle_data}()\) with the character as argument. If \(\text{ref}\) is invalid or out of range, the method unknown_charref(\text{ref}) is called instead.

- \(\text{handle_entityref}(\text{ref})\)
  - This method is called to process an entity reference of the form \('&\#ref;'\) where \(\text{ref}\) is an alphabetic entity reference. It looks for \(\text{ref}\) in the instance (or class) variable \(\text{entitydefs}\) which should give the entity’s translation. If a translation is found, it calls the method \(\text{handle_data}()\) with the translation; otherwise, it calls the method unknown_entityref(\text{ref}).

- \(\text{handle_data}(\text{data})\)
  - This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

- \(\text{unknown_starttag}(\text{tag}, \text{attributes})\)
  - This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing. The \(\text{attributes}\) argument is a list of \((\text{name}, \text{value})\) pairs containing the attributes found inside the tag’s \(<\text{tag}>...<\text{tag}>\) brackets. The \text{name} has been translated to lower case and double quotes and backslashes in the \text{value} have been interpreting.

For instance, for the tag \('<A \text{HREF}="\text{http://www.cwi.nl/}"\), this method would be called as unknown_starttag(’a’, [[’href’, ’http://www.cwi.nl/’]]).

- \(\text{unknown_endtag}(\text{tag})\)
  - This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

- \(\text{unknown_charref}(\text{ref})\)
  - This method is called to process an unknown character reference. It is intended to be overridden by a derived class; the base class implementation does nothing.
unknown_entityref( ref )

This method is called to process an unknown entity reference. It is intended to be overridden by a derived class, the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the tag occurring in method names must be in lower case:

- **start_tag(attributes)**
  - This method is called to process an opening tag tag. It has preference over do_tag(). The attributes argument has the same meaning as described for unknown_tag() above.

- **do_tag(attributes)**
  - This method is called to process an opening tag tag that does not come with a matching closing tag. The attributes argument has the same meaning as described for unknown_tag() above.

- **end_tag()**
  - This method is called to process a closing tag tag.

Note that the parser maintains a stack of opening tags for which no matching closing tag has been found yet. Only tags processed by start_tag() are pushed on this stack. Definition of an end_tag() method is optional for these tags. For tags processed by do_tag() or by unknown_tag(), no end_tag() method must be defined.

### 11.10 Standard Module rfc822

This module defines a class, *Message*, which represents a collection of "email headers" as defined by the Internet standard RFC 822. It is used in various contexts, usually to read such headers from a file.

A *Message* instance is instantiated with an open file object as parameter. Instantiation reads headers from the file up to a blank line and stores them in the instance; after instantiation, the file is positioned directly after the blank line that terminates the headers.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. `m['From']`, `m['from']` and `m['FROM']` all yield the same result.

#### 11.10.1 Message Objects

A *Message* instance has the following methods:

- **rewindbody()**
  - Seek to the start of the message body. This only works if the file object is seekable.

- **getallmatchingheaders(name)**
  - Return a list of lines consisting of all headers matching name, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches name.

- **getfirstmatchingheader(name)**
  - Return a list of lines comprising the first header matching name, and its continuation line(s), if any. Return None if there is no header matching name.

- **getrawheader(name)**
  - Return a single string consisting of the text after the colon in the first header matching name. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation lines() were present. Return None if there is no header matching name.

- **getheader(name)**
  - Like getrawheader(name), but strip leading and trailing whitespace (but not internal whitespace).

- **getaddr(name)**
  - Return a pair (full name, email address) parsed from the string returned by getheader(name). If no header matching name exists, return None; otherwise both the full name and the address are (possibly empty) strings.

  Example: If m's first From header contains the string 'jack@cwi.nl (Jack Jansen)', then m.getaddr('From') will yield the pair ('Jack Jansen', 'jack@cwi.nl'). If the header contained 'Jack Jansen <jack@cwi.nl>' instead, it would yield the exact same result.

- **getaddrlist(name)**
  - This is similar to getaddr(list), but parses a header containing a list of email addresses (e.g. a To header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching name, return an empty list.

- **getdate(name)**
  - Return a header using getheader and parse it as a 9-tuple compatible with time.mktime(). If there is no header matching name, or it is unparsable, return None.

  Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

- ** kullas_kids()**
  - Define this method if you want it to be called on a particular body object.

  This is a dummy method used for the purpose of demonstrating that methods can be called on body objects.

Message instances also support a read-only mapping interface. In particular: m[name] is the same as m.getheader(name); and m.getheaders() and m.items() (and m.keys() and m.values() and m.items() acts as expected) (and consistently).

Finally, *Message* instances have two public instance variables:

- **headers**
  - A list containing the entire set of header lines, in the order in which they were read. Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

- **fp**
  - The file object passed at instantiation time.

### 11.11 Standard Module mimetools

This module defines a subclass of the class *rfc822.Message* and a number of utility functions that are useful for the manipulation for MIME style multipart or encoded message.

It defines the following items:

- **Message(fp)**
  - Returns a new instance of the mimetools.Message class. This is a subclass of the rfc822.Message class, with some additional methods (see below).

- **choose_boundary()**
  - Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form "hostpaddl_ads.pul.timestamp.random".

- **getnextheader()**
  - Return a header as parsed from the string returned by getallmatchingheaders(). If no header matching name exists, return None; otherwise both the full name and the address are (possibly empty) strings.

- **getfirstheader()**
  - Similar to getnextheader, but stops after the first matching header.

- **getnextline()**
  - Return the next line of the message stream.

- **getallheaders()**
  - Similar to getallmatchingheaders, but without the limit to only one header.

- **getalllines()**
  - Return a list of all lines of the message stream.

- **getallmultipart()**
  - Return a list of all multipart objects contained in the message.

- **getallencoded()**
  - Return a list of all encoded objects contained in the message.

- **getallextended()**
  - Return a list of all extended objects contained in the message.

- **getallmain()**
  - Return a list of all main objects contained in the message.

- **getallother()**
  - Return a list of all other objects contained in the message.

- **getfirstmatchingheader()**
  - Similar to getfirstheader, but stops after the first matching header.

- **getfirstline()**
  - Return the first line of the message.

- **getfirstpart()**
  - Return the first part of the message.

- **getfirstencoded()**
  - Return the first encoded object contained in the message.

- **getfirstextended()**
  - Return the first extended object contained in the message.

- **getfirstmain()**
  - Return the first main object contained in the message.

- **getfirstother()**
  - Return the first other object contained in the message.

- **getheaders()**
  - Return a list of all headers of the message.

- **getlines()**
  - Return a list of all lines of the message.

- **getparts()**
  - Return a list of all parts of the message.

- **getencoded()**
  - Return a list of all encoded objects contained in the message.

- **getextended()**
  - Return a list of all extended objects contained in the message.

- **getmain()**
  - Return a list of all main objects contained in the message.

- **getother()**
  - Return a list of all other objects contained in the message.

- **getboundary()**
  - Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form "hostpaddl_ads.pul.timestamp.random".

- **getpadder()**
  - Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form "hostpaddl_ads.pul.timestamp.random".

- **gettimestamp()**
  - Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form "hostpaddl_ads.pul.timestamp.random".

- **getrandom()**
  - Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form "hostpaddl_ads.pul.timestamp.random".
11.11.1 Additional Methods of Message objects

The `mimetools.Message` class defines the following methods in addition to the `rfc822.Message` class:

- `getplist()`
  - Return the parameter list of the `Content-type` header. This is a list of strings. For parameters of the form `key=value`, `key` is converted to lower case but `value` is not. For example, if the message contains the header `Content-type: text/html; spam=1; Spam=2; Spam` then `getplist()` will return the Python list `['spam=1', 'spam=2', 'Spam']`.

- `getparam(name)`
  - Return the value of the first parameter (as returned by `getplist()` of the form `name=value`) for the given `name`. If `value` is surrounded by quotes of the form `<...>` or `"..."`, these are removed.

- `getencoding()`
  - Return the encoding specified in the `Content-transfer-encoding` message header. If no such header exists, return "7bit". The encoding is converted to lower case.

- `gettype()`
  - Return the message type (of the form `type/subtype`) as specified in the `Content-type` header. If no such header exists, return "text/plain". The type is converted to lower case.

- `getmaintype()`
  - Return the main type as specified in the `Content-type` header. If no such header exists, return "text". The main type is converted to lower case.

- `getsubtype()`
  - Return the subtype as specified in the `Content-type` header. If no such header exists, return "plain". The subtype is converted to lower case.

Chapter 12
Multimedia Services

The modules described in this chapter implement various algorithms and interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

- `audioop` — Manipulate raw audio data.
- `imageop` — Manipulate raw image data.
- `aifc` — Read and write audio files in AIFF or AIFC format.
- `jpeg` — Read and write image files in compressed JPEG format.
- `rgbimg` — Read and write image files in "SGI RGB" format (the module is not SGI specific though!)

12.1 Built-in Module `audioop`

The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the `al` and `sunaudiodev` modules. All scalar items are integers, unless specified otherwise.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

- `error`
  - This exception is raised on all errors, such as unknown number of bytes per sample, etc.

- `add(fragment1, fragment2, width)`
  - Return a fragment which is the addition of the two samples passed as parameters. `width` is the sample width in bytes, either 1, 2 or 4. Both fragments should have the same length.

- `adpcm2lin(adpcmfragment, width, state)`
  - Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm` for details on ADPCM coding. Return a tuple `(sample, newstate)` where the sample has the width specified in `width`.

- `adpcm32lin(adpcmfragment, width, state)`
  - Decode an alternative 3-bit ADPCM code. See `lin2adpcm3` for details.

- `avg(fragment, width)`
  - Return the average over all samples in the fragment.
avgpp(\text{fragment}, width)
Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

bias(\text{fragment}, width, bias)
Return a fragment that is the original fragment with a bias added to each sample.

cross(\text{fragment}, width)
Return the number of zero crossings in the fragment passed as an argument.

findfactor(\text{fragment}, reference)
Return a factor \( F \) such that \( \text{rms}(\text{add}(\text{fragment}, \text{mul}(\text{reference}, -F))) \) is minimal, i.e. return the factor with which you should multiply \text{reference} to make it match as well as possible to \text{fragment}. The fragments should both contain 2-byte samples. The time taken by this routine is proportional to \( \text{len(fragment)} \).

findmax(\text{fragment}, length)
Search \text{fragment} for a slice of length \( \text{length} \) samples (not bytes!) with maximum energy, i.e., return \( i \) for which \( \text{rms}(\text{fragment}[i:2+(i+\text{length})*2]) \) is maximal. The fragments should both contain 2-byte samples. The routine takes time proportional to \( \text{len(fragment)} \).

getsample(\text{fragment}, width, index)
Return the value of sample \text{index} from the fragment.

lin2adpcm(\text{fragment}, width, state)
Convert samples to 4 bit Intel/VDVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/VDVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

State is a tuple containing the state of the coder. The coder returns a tuple \((\text{adpcmfrag}, \text{newstate})\), and the \text{newstate} should be passed to the next call of \text{lin2adpcm}. In the initial call None can be passed as the state. \text{adpcmfrag} is the ADPCM coded fragment packed 2-4 bit values per byte.

lin2adpcm3(\text{fragment}, width, state)
This is an alternative ADPCM coder that uses only 3 bits per sample. It is not compatible with the Intel/VDVI ADPCM coder and its output is not packed (due to laziness on the side of the author). Its use is discouraged.

lin2ulaw(\text{fragment}, width)
Convert samples in the audio fragment to U-LAW encoding and return this as a Python string. U-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

minmax(\text{fragment}, width)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

max(\text{fragment}, width)
Return the maximum of the absolute value of all samples in a fragment.

mul(\text{fragment}, width, factor)
Return the maximum peak-peak value in the sound fragment.

mul_stereo(sample, width, lfactor, rfactor):
\[
def \text{mul_stereo}(\text{sample}, \text{width}, \text{lfactor}, \text{rfactor}): 
\begin{align*} 
\text{lsample} &= \text{audioop.tomonono}(\text{sample}, \text{width}, 1, 0) 
\text{rsample} &= \text{audioop.tomonono}(\text{sample}, \text{width}, 0, 1) 
\text{lsample} &= \text{audioop.mul}(\text{sample}, \text{width}, \text{lfactor}) 
\text{rsample} &= \text{audioop.mul}(\text{sample}, \text{width}, \text{rfactor}) 
\text{lsample} &= \text{audioop.tostereo}(\text{lsample}, \text{width}, 1, 0) 
\text{rsample} &= \text{audioop.tostereo}(\text{rsample}, \text{width}, 0, 1) 
\text{return audioop.add}(\text{lsample}, \text{rsample}, \text{width}) 
\end{align*} 
\]

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the initial state (the one you passed to \text{lin2adpcm}) along to the decoder, not the final state (as returned by the coder). If you want to use \text{struct} to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against each other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The \text{find...} routines might look a bit funny at first sight. They are primarily meant to do echo cancellation.
A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 880)  # one tenth second
    in_test = outputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    # out_test)
    prefill = '\0'*((pos+ipos)*2)
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefilt + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)

12.2 Built-in Module imageop

The imageop module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixel stored in Python strings. This is the same format as used by gl.lirectwrite and the imgfile module.

The module defines the following variables and functions:

- **error**: This exception is raised on all errors, such as unknown number of bits per pixel, etc.

- **crop** (image, psize, width, height, x0, y0, x1, y1)
  - Return the selected part of image, which should by width by height in size and consist of pixels of psize bytes. x0, y0, x1 and y1 are like the lirectread parameters, ie the boundary is included in the new image. The new boundaries need not be inside the picture. Pixels that fall outside the old image will have their value set to zero. If x0 is bigger than x1 the new image is mirrored. The same holds for the y coordinates.

- **scale** (image, psize, width, height, newwidth, newheight)
  - Return image scaled to size newwidth by newheight. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

- **tovideo** (image, psize, width, height)
  - Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

- **grey2monochrome** (image, width, height, threshold)
  - Convert a 8-bit deep greyscale image to a 1-bit deep image by thresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to **mono2grey**.

- **dither2mono** (image, width, height)
  - Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

- **mono2grey** (image, width, height, p0, p1)
  - Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value p0 on output and all one-value input pixels get value p1 on output. To convert a monochrome black-and-white image to greyscale pass the values 0 and 255 respectively.

- **grey2gray4** (image, width, height)
  - Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

- **grey2gray2** (image, width, height)
  - Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

- **dither2gray2** (image, width, height)
  - Convert an 8-bit greyscale image to a 2-bit greyscale image with dithering. As for dither2mono, the dithering algorithm is currently very simple.

- **grey2gray** (image, width, height)
  - Convert a 4-bit greyscale image to an 8-bit greyscale image.

- **grey2gray2** (image, width, height)
  - Convert a 2-bit greyscale image to an 8-bit greyscale image.

12.3 Standard Module aifc

This module provides support for reading and writing AIFF and AIFF-C® files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a®le. AIFF-C® is a newer version of the format that includes the ability to compress the audio data.

Audio®les have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indica©e the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of nchannels * samplesize * framerate bytes, and a second's worth of audio consists of nchannels * samplesize * framerate * bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes, ie 176,400 bytes.

Module aifc defines the following functions:

- **open**(file, mode)
  - Open a AIFF or AIFF-C® file and return an object instance with methods that are described below. The argument file is either a string naming a file or a file object. The mode is either the string `r` when the file must be opened for reading, or `w` when the file must be opened for writing. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use writeframesraw and setnframes.

Objects returned by aifc.open() when a file is opened for reading have the following methods:

- **getnchannels()**
  - Return the number of audio channels (1 for mono, 2 for stereo).

- **getsampwidth()**
  - Return the size in bytes of individual samples.

- **getframerate()**
  - Return the sampling rate (number of audio frames per second).

- **getnframes()**
  - Return the number of audio frames in the file.
getcomptype()  
Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is `"NONE"`.

getcomppname()  
Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `"not compressed"`.

getparams()  
Return a tuple consisting of all of the above values in the above order.

getmarkers()  
Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

getmark(id)  
Return the tuple as described in getmarkers for the mark with the given id.

readframes(nframes)  
Read and return the next nframes frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

rewind()  
Rewind the read pointer. The next readframes will start from the beginning.

setpos(pos)  
Seek to the specified frame number.

tell()  
Return the current frame number.

close()  
Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by aiff.open() when a file is opened for writing have all the above methods, except for readframes and setpos. In addition the following methods exist. The get methods can only be called after the corresponding set methods have been called. Before the first writeframes or writeframexraw, all parameters except for the number of frames must be filled in.

aiff()  
Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in `.aiff` in which case the default is an AIFF file.

aiffc()  
Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in `.aiff` in which case the default is an AIFF file.

setnchannels(nchannels)  
Specify the number of channels in the audio file.

setnframes(nframes)  
Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

setcomptype(type, name)  
Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

setparams(nchannels, sampwidth, framerate, comptype, compname)  
Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This set method call as argument to setparams.

setmark(id, pos, name)  
Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called any time before close.

tell()  
Return the current write position in the output file. Useful in combination with setmark.

writeframes(data)  
Write data to the output file. This method can only be called after the audio file parameters have been set.

writeframexraw(data)  
Like writeframes, except that the header of the audio file is not updated.

close()  
Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

12.4 Built-in Module jpeg

The module jpeg provides access to the jpeg compressor and decompressor written by the Independent JPEG Group. JPEG is a (draft?) standard for compressing pictures. For details on jpeg or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

The jpeg module defines these functions:

compress(data, w, h, b)  
Treat data as a pixmap of width w and height h, with b bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that lrectread return data can immediately be passed to compress. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. Compress returns a string that contains the compressed picture, in JFIF format.

decompress(data)  
Data is a string containing a picture in JFIF format. It returns a tuple (data, width, height, bytesperpixel). Again, the data is suitable to pass to lrectwrite.

setoption(name, value)  
Set various options. Subsequent compress and decompress calls will use these options. The following options are available:

'forcegray' Force output to be grayscale, even if input is RGB.

'quality' Set the quality of the compressed image to a value between 0 and 100 (default is 75). Compress only.

'optimize' Perform Huffman table optimization. Takes longer, but results in smaller compressed image. Compress only.
Perform inter-block smoothing on uncompressed image. Only useful for low-quality images. Decompress only.

Compress and uncompress raise the error `jpeg.error` in case of errors.

### 12.5 Built-in Module rgbimg

The `rgbimg` module allows Python programs to access SGI `imglib` image files (also known as `.rgb` files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

- `error`:
  This exception is raised on all errors, such as unsupported file type, etc.

- `sizeofimage(file)`:
  This function returns a tuple \((x, y)\) where \(x\) and \(y\) are the size of the image in pixels. Only 4 byte RGBA pixels, 3 byte RGB pixels, and 1 byte greyscale pixels are currently supported.

- `longimageupdate(file)`:
  This function reads and decodes the image on the specified file, and returns it as a Python string. The string has 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to `gl.lrectwrite`, for instance.

- `longstoinmage(data, x, y, z, file)`:
  This function writes the RGBA data in `data` to image file `file`. \(x\) and \(y\) give the size of the image. \(z\) is 1 if the saved image should be 1 byte greyscale, 3 if the saved image should be 3 byte RGB data, or 4 if the saved image should be 4 byte RGBA data. The input data always contains 4 bytes per pixel. These are the formats returned by `gl.lrectread`.

- `ttob(flag)`:
  This function sets a global flag which defines whether the scanlines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.

### Chapter 13

Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here’s an overview:

- `md5` — RSA’s MD5 message digest algorithm.
- `mpz` — Interface to the GNU MP library for arbitrary precision arithmetic.
- `rotor` — Enigma-like encryption and decryption.

Hardcore cyberpunks will probably find the Python Cryptography Kit of further interest; the package adds built-in modules for DES and IDEA encryption, and provides a Python module for reading and decrypting PGP files. The Python Cryptography Kit is not distributed with Python but available separately. See the URL `http://www.cs.mcgill.ca/~fnord/crypt.html` for more information.

#### 13.1 Built-in Module md5

This module implements the interface to RSA’s MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use the `md5.new()` to create an `md5` object. You can now feed this object with arbitrary strings using the `update()` method, and at any point you can ask for the digest (a strong kind of 128-bit checksum, a.k.a. “fingerprint”) of the concatenation of the strings fed to it so far using the `digest()` method.

For example, to obtain the digest of the string “Nobody inspects the spammish repetition”:

```python
>>> import md5
>>> m = md5.new()
>>> m.update("Nobody inspects")
>>> m.update("the spammish repetition")
>>> m.digest()
'\x273d\x234\x203\x335\x036\x245\x311\x336\x311\x241\x215\x360\x377\x351'
```

More condensed:

```python
>>> md5.new("Nobody inspects the spammish repetition").digest()
'\x273d\x234\x203\x335\x036\x245\x311\x336\x311\x241\x215\x360\x377\x351'
```

new([arg])
Return a new mpz object. If arg is present, the method call update(arg) is made.

md5([arg])
For backward compatibility reasons, this is an alternative name for the new() function.

An md5 object has the following methods:
update(arg)
Update the md5 object with the string arg. Repeated calls are equivalent to a single call with the concatenation of all the arguments, i.e., m.update(a); m.update(b) is equivalent to m.update(a+b).
digest()
Return the digest of the strings passed to the update() method so far. This is an 8-byte string which may contain non-ASCII characters, including null bytes.
copy()
Return a copy ("clone") of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

13.2 Built-in Module mpz

This module implements the interface to part of the GNU MP library. This library contains arbitrary precision integer and rational number arithmetic routines. Only the interfaces to the integer (mpz, _...) routines are provided. If not stated otherwise, the description in the GNU MP documentation can be applied.

In general, mpz-numbers can be used just like other standard Python numbers, e.g., you can use the built-in operators like +, *, etc., as well as the standard built-in functions like abs, int, ..., divmod, pow. Please note: the bitwise-xor operation has been implemented as a bunch of ands, inverts and ors, because the library lacks an mpz_xor function, and I didn't need one.

You create an mpz-number by calling the function called mpz (see below for an exact description). An mpz-number is printed like this: value(mpz).value

mpz(value)
Create a new mpz-number. value can be an integer, a long, another mpz-number, or even a string. If it is a string, it is interpreted as an array of radix-256 digits, least significant digit first, resulting in a positive mpz-number. See also the binary method, described below.

A number of extra functions are defined in this module. Non mpz-arguments are converted to mpz-values first, and the functions return mpz-numbers.

elem(base, exponent, modulus)
Return pow(base, exponent) % modulus. If exponent == 0, return mpz(1). In contrast to the C-library function, this version can handle negative exponents.
gcd(op1, op2)
Return the greatest common divisor of op1 and op2.
gecdtext(a, b)
Return a tuple (g, s, t), such that a*s + b*t == g == gcd(a, b).
sqrt(op)
Return the square root of op. The result is rounded towards zero.
sqrtrem(op)
Return a tuple (root, remainder), such that root*root + remainder == op.
divm(numerator, denominator, modulus)
Returns a number q, such that q * denominator % modulus == numerator. One could also implement this function in Python, using gcdext.

An mpz-number has one method:
binary()
Convert this mpz-number to a binary string, where the number has been stored as an array of radix-256 digits, least significant digit first.

The mpz-number must have a value greater than or equal to zero, otherwise a ValueError exception will be raised.

13.3 Built-in Module rotor

This module implements a rotor-based encryption algorithm, contributed by Lance Ellingham. The design is derived from the Enigma device, a machine used during World War II to encipher messages. A rotor is simply a permutation. For example, if the character 'A' is the origin of the rotor, then a given rotor might map 'A' to 'L', 'B' to 'Z', 'C' to 'G', and so on. To encrypt, we choose several different rotors, and set the origins of the rotors to known positions; their initial position is the ciphering key. To encrypt a character, we permute the original character by the first rotor, and then apply the second rotor's permutation to the result. We continue until we've applied all the rotors; the resulting character is our ciphertext. We then change the origin of the final rotor by one position, from 'A' to 'B'; if the final rotor has made a complete revolution, then we rotate the next-to-last rotor by one position, and apply the same procedure recursively. In other words, after encrypting one character, we advance the rotors in the same fashion as a car's odometer. Decoding works in the same way, except we reverse the permutations and apply them in the opposite order.

The available functions in this module are:

newrotor(key, numrotors)
Return a new rotor object. key is a string containing the encryption key for the object; it can contain arbitrary binary data. The key will be used to randomly generate the rotor permutations and their initial positions. numrotors is the number of rotor permutations in the returned object; if it is omitted, a default value of 6 will be used.

Rotor objects have the following methods:

setkey()
Reset the rotor to its initial state.

encrypt(plaintext)
Reset the rotor object to its initial state and encrypt plaintext, returning a string containing the ciphertext. The ciphertext is always the same length as the original plaintext.

decrypt(ciphertext)
Decrypt the rotor object to its initial state and decrypt ciphertext, returning a string containing the ciphertext. The plaintext string will always be the same length as the ciphertext.

decryptmore(ciphertext)
Decrypt ciphertext without resetting the rotor object, and return a string containing the ciphertext.

An example usage:

119
The module’s code is not an exact simulation of the original Enigma device; it implements the rotor encryption scheme differently from the original. The most important difference is that in the original Enigma, there were only 5 or 6 different rotors in existence, and they were applied twice to each character; the cipher key was the order in which they were placed in the machine. The Python rotor module uses the supplied key to initialize a random number generator; the rotor permutations and their initial positions are then randomly generated. The original device only enciphered the letters of the alphabet, while this module can handle any 8-bit binary data; it also produces binary output. This module can also operate with an arbitrary number of rotors.

The original Enigma cipher was broken in 1944. The version implemented here is probably a good deal more difficult to crack (especially if you use many rotors), but it won’t be impossible for a truly skillful and determined attacker to break the cipher. So if you want to keep the NSA out of your files, this rotor cipher may well be unsafe, but for discouraging casual snooping through your files, it will probably be just fine, and may be somewhat safer than using the Unix `crypt` command.

Chapter 14

Macintosh Specific Services

The modules in this chapter are available on the Apple Macintosh only.

14.1 Built-in Module mac

This module provides a subset of the operating system dependent functionality provided by the optional built-in module `posix`. It is best accessed through the more portable standard module `os`.

The following functions are available in this module: `chdir`, `getcwd`, `listdir`, `mkdir`, `rename`, `rmdir`, `stat`, `sync`, `unlink`, as well as the exception `error`.

14.2 Standard Module macpath

This module provides a subset of the pathname manipulation functions available from the optional standard module `posixpath`. It is best accessed through the more portable standard module `os`, as `os.path`.

The following functions are available in this module: `isabs`, `isfile`, `isdir`, `isexists`, `exists`.

14.3 Built-in Module ctb

This module provides a partial interface to the Macintosh Communications Toolbox. Currently, only Connection Manager tools are supported. It may not be available in all Mac Python versions.

`error`

The exception raised on errors.

`cmData`

`cmCntl`

`cmAttn`

Flags for the `channel` argument of the `Read` and `Write` methods.

`cmFlagEOM`

End-of-message flag for `Read` and `Write`. 
choose*

Values returned by Choose.

cmStatus*

Bits in the status as returned by Status.

available()

Return 1 if the communication toolbox is available, zero otherwise.

CMNew(name, sizes)

Create a connection object using the connection tool named name. sizes is a 6-tuple giving buffer sizes for data in, data out, control in, control out, attention in and attention out. Alternatively, passing None will result in default buffer sizes.

14.3.1 connection object

For all connection methods that take a timeout argument, a value of -1 is indefinite, meaning that the command runs to completion.

callback

If this member is set to a value other than None it should point to a function accepting a single argument (the connection object). This will make all connection object methods work asynchronously, with the callback routine being called upon completion.

Note: for reasons beyond my understanding the callback routine is currently never called. You are advised against using asynchronous calls for the time being.

Open(timeout)

Open an outgoing connection, waiting at most timeout seconds for the connection to be established.

Listen(timeout)

Wait for an incoming connection. Stop waiting after timeout seconds. This call is only meaningful to some tools.

accept(yesno)

Accept (when yesno is non-zero) or reject an incoming call after Listen returned.

close(timeout, now)

Close a connection. When now is zero, the close is orderly (i.e. outstanding output is flushed, etc.) with a timeout of timeout seconds. When now is non-zero the close is immediate, discarding output.

read(len, chan, timeout)

Read len bytes, or until timeout seconds have passed, from the channel chan (which is one of cmData, cmOrd or cmAttn). Return a 2-tuple: the data read and the end-of-message flag.

write(buf, chan, timeout, eom)

Write buf to channel chan, aborting after timeout seconds. When eom has the value cmFlagsEOM an end-of-message indicator will be written after the data (if this concept has a meaning for this communication tool). The method returns the number of bytes written.

Status()

Return connection status as the 2-tuple (sizes, flags). sizes is a 6-tuple giving the actual buffer sizes used (see CMNew), flags is a set of bits describing the state of the connection.

GetConfig()

Return the configuration string of the communication tool. These configuration strings are tool-dependent, but usually easily parsed and modified.

SetConfig(str)

Set the configuration string for the tool. The strings are parsed left-to-right, with later values taking precedence. This means individual configuration parameters can be modified by simply appending something like ‘baud 4800’ to the end of the string returned by GetConfig and passing that to this method. The method returns the number of characters actually parsed by the tool before it encountered an error (or completed successfully).

Choose()

Present the user with a dialog to choose a communication tool and configure it. If there is an outstanding connection some choices (like selecting a different tool) may cause the connection to be aborted. The method returns the number of characters actually parsed by the tool before it encountered an error (or completed successfully).

Idle()

Give the tool a chance to use the processor. You should call this method regularly.

Abort()

Abort an outstanding asynchronous Open or Listen.

Reset()

Reset a connection. Exact meaning depends on the tool.

Break(length)

Send a break. Whether this means anything, what it means and interpretation of the length parameter depend on the tool in use.

14.4 Built-in Module macconsole

This module is available on the Macintosh, provided Python has been built using the Think C compiler. It provides an interface to the Think console package, with which basic text windows can be created.

options

An object allowing you to set various options when creating windows, see below.

C_ECHO

C_NOECHO

C_CBR

C_RAW

Options for the setmode method. C_ECHO and C_CBR enable character echo, the other two disable it, C_ECHO and C_NOECHO enable line-oriented input (erase/kill processing, etc).

copen()

Open a new console window. Return a console window object.

fopen(fp)

Return the console window object corresponding with the given file object. fp should be one of sys.stdin, sys.stdout or sys.stderr.

14.4.1 macconsole options object

These options are examined when a window is created:

top

left

The origin of the window.

nwmb
ncols
The size of the window.
txFont
txSize
txStyle
The font, fontsize and fontstyle to be used in the window.
title
The title of the window.
pause_atexit
If set non-zero, the window will wait for user action before closing.

14.4.2 console window object
file
The file object corresponding to this console window. If the file is buffered, you should call file.flush() between write() and read() calls.
setmode(mode)
Set the input mode of the console to C_ECHO, etc.
settabs(n)
Set the tabsize to n spaces.
cleos()
Clear to end-of-screen.
cleol()
Clear to end-of-line.
inverse(onoff)
Enable inverse-video mode: characters with the high bit set are displayed in inverse video (this disables the upper half of a non-ASCII character set).
gotoxy(x,y)
Set the cursor to position (x,y).
hide()
Hide the window, remembering the contents.
show()
Show the window again.
echo2printer()
Copy everything written to the window to the printer as well.

14.5 Built-in Module macdnr
This module provides an interface to the Macintosh Domain Name Resolver. It is usually used in conjunction with the mactcp module, to map hostnames to IP-addresses. It may not be available in all Mac Python versions.
The macdnr module defines the following functions:
Open( )
Open the domain name resolver extension. If filename is given it should be the pathname of the extension, otherwise a default is used. Normally, this call is not needed since the other calls will open the extension automatically.
Close()
Close the resolver extension. Again, not needed for normal use.
StrToAddr(hostname)
Look up the IP address for hostname. This call returns a dnr result object of the "address" variation.
AddrToName(addr)
Do a reverse lookup on the 32-bit integer IP-address addr. Returns a dnr result object of the "address" variation.
AddrToStr(addr)
Convert the 32-bit integer IP-address addr to a dotted-decimal string. Returns the string.
HInfo(hostname)
Query the nameservers for a Hinfo record for host hostname. These records contain hardware and software information about the machine in question (if they are available in the first place). Returns a dnr result object of the "hinfo" variety.
MXInfo(domain)
Query the nameservers for a mail exchanger for domain. This is the hostname of a host willing to accept SMTP mail for the given domain. Returns a dnr result object of the "mx" variety.

14.5.1 dnr result object
Since the DNR calls all execute asynchronously you do not get the results back immediately. Instead, you get a dnr result object. You can check this object to see whether the query is complete, and access its attributes to obtain the information when it is.
Alternatively, you can also reference the result attributes directly; this will result in an implicit wait for the query to complete.
The rtnCode and cname attributes are always available, the others depend on the type of query (address, hinfo or mx).
wait()
Wait for the query to complete.
isdone()
Return 1 if the query is complete.
rtnCode
The error code returned by the query.
cname
The canonical name of the host that was queried.

ip0
ip1
ip2
ip3
At most four integer IP addresses for this host. Unused entries are zero. Valid only for address queries.
cpuType
osType
Textual strings giving the machine type and OS name. Valid for hinfo queries.
The name of a mail-exchanger host. Valid for mx queries.

preference
The preference of this mx record. Not too useful, since the Macintosh will only return a single mx record.
Mx queries only.
The simplest way to use the module to convert names to dotted-decimal strings, without worrying about idle
time, etc.

>>> def gethostname(name):
...     import macdnr
...     dnrr = macdnr.StrToAddr(name)
...     return macdnr.AddrToStr(dnrr.ip0)

14.6 Built-in Module macfs

This module provides access to macintosh FSSpec handling, the Alias Manager, finder aliases and the Standard
File package.

Whenever a function or method expects a file argument, this argument can be one of three things: (1) a full
or partial Macintosh pathname, (2) an FSSpec object or (3) a 3-tuple (wdRefNum, parID, name) as
described in Inside Mac VI. A description of aliases and the standard file package can also be found there.

FSSpec(file)
Create an FSSpec object for the specified file.

RawFSSpec(data)
Create an FSSpec object given the raw data for the C structure for the FSSpec as a string. This is mainly
useful if you have obtained an FSSpec structure over a network.

RawAlias(data)
Create an Alias object given the raw data for the C structure for the alias as a string. This is mainly useful
if you have obtained an Alias structure over a network.

ResolveAliasFile(file)
Resolve an alias file. Returns a 3-tuple (fsspec, isfolder, aliased) where fsspec is the resulting
FSSpec object, isfolder is true if fsspec points to a folder and aliased is true if the file was an alias in the
first place (otherwise the FSSpec object for the file itself is returned).

StandardGetFile(type, ...)
Present the user with a standard "open input file" dialog. Optionally, you can pass up to four 4-char
file types to limit the files the user can choose from. The function returns an FSSpec object and a flag
indicating that the user completed the dialog without cancelling.

StandardPutFile(prompt, default)
Present the user with a standard "open output file" dialog. prompt is the prompt string, and the optional
default argument initializes the output file name. The function returns an FSSpec object and a flag
indicating that the user completed the dialog without cancelling.

GetDirectory()
Present the user with a non-standard "select a directory" dialog. Return an FSSpec object and a success-

14.6.1 FSSpec objects

data
The raw data from the FSSpec object, suitable for passing to other applications, for instance.
as_pathname()
Return the full pathname of the file described by the FSSpec object.
as_tuple()
Return the (wdRefNum, parID, name) tuple of the file described by the FSSpec object.

NewAlias(file)
Create an Alias object pointing to the file described by this FSSpec. If the optional file parameter is
present the alias will be relative to that file, otherwise it will be absolute.

NewAliasMinimal()
Create a minimal alias pointing to this file.

GetCreatorType()
Return the 4-char creator and type of the file.

14.7 Built-in Module mactcp

This module provides an interface to the Macintosh TCP/IP driver MacTCP. There is an accompanying module
macnx which provides an interface to the name-server (allowing you to translate hostnames to ip-addresses),
a module MACTCP which has symbolic names for constants constants used by MacTCP and a wrapper module
socket which mimics the UNIX socket interface (as far as possible). It may not be available in all Mac Python
versions.

A complete description of the MacTCP interface can be found in the Apple MacTCP API documentation.

MTU()
Return the Maximum Transmit Unit (the packet size) of the network interface.

IPAddr()
Return the 32-bit integer IP address of the network interface.
NetMask()
Return the 32-bit integer network mask of the interface.

TCPCreate(size)
Create a TCP Stream object. size is the size of the receive buffer. 4096 is suggested by various sources.

UDPCreate(size, port)
Create a UDP stream object. size is the size of the receive buffer (and, hence, the size of the biggest
datagram you can receive on this port). port is the UDP port number you want to receive datagrams on,
a value of zero will make MacTCP select a free port.

14.7.1 TCP Stream Objects

asr
When set to a value different than None this should point to a function with two integer parameters: an
event code and a detail. This function will be called upon network-generated events such as urgent data
arrival. In addition, it is called with eventcode MacTCP.PassiveOpenDone when a PassiveOpen
completes. This is a Python addition to the MacTCP semantics. It is safe to do further calls from the
asr.

PassiveOpen(port)
Wait for an incoming connection on TCP port port (zero makes the system pick a free port). The call
returns immediately, and you should use wait to wait for completion. You should not issue any method
calls other than wait, isdone or GetSockName before the call completes.

wait()
Wait for PassiveOpen to complete.

isdone()
Return 1 if a PassiveOpen has completed.

GetSockName()
Return the TCP address of this side of a connection as a 2-tuple (host, port), both integers.

ActiveOpen(lport, host, rport)
Open an outgoing connection to TCP address (host, rport). Use local port lport (zero makes the
system pick a free port). This call blocks until the connection has been established.

Send(buf, push, urgent)
Send data buf over the connection. Push and urgent are flags as specified by the TCP standard.

Rcv(timeout)
Receive data. The call returns when timeout seconds have passed or when (according to the MacTCP
documentation) “a reasonable amount of data has been received”. The return value is a 3-tuple
(data, urgent, mark). If urgent data is outstanding Rcv will always return that before looking
at any normal data. The first call returning urgent data will have the urgent flag set, the last will have the
mark flag set.

Close()
Tell MacTCP that no more data will be transmitted on this connection. The call returns when all data has
been acknowledged by the receiving side.

Abort()
Forcibly close both sides of a connection, ignoring outstanding data.

Status()
Return a TCP status object for this stream giving the current status (see below).

14.7.2 TCP Status Objects

This object has no methods, only some members holding information on the connection. A complete description
of all fields in this object can be found in the Apple documentation. The most interesting ones are:

localHost
localPort
remoteHost
remotePort
The integer IP-addresses and port numbers of both endpoints of the connection.

sendWindow
The current window size.

amtUnackedData
The number of bytes sent but not yet acknowledged. sendWindow - amtUnackedData is what
you can pass to Send without blocking.

amtUnreadData
The number of bytes received but not yet read (what you can Rcv without blocking).

14.7.3 UDP Stream Objects

Note that, unlike the name suggests, there is nothing stream-like about UDP.

asr
The asynchronous service routine to be called on events such as datagram arrival without outstanding
Read call. The asr has a single argument, the event code.

port
A read-only member giving the port number of this UDP stream.

Read(timeout)
Read a datagram, waiting at most timeout seconds (−1 is infinite). Return the data.

Write(host, port, buf)
Send buf as a datagram to IP-address host, port port.

14.8 Built-in Module macspeech

This module provides an interface to the Macintosh Speech Manager, allowing you to let the Macintosh utter
phrases. You need a version of the speech manager extension (version 1 and 2 have been tested) in your
Extensions folder for this to work. The module does not provide full access to all features of the Speech
Manager yet. It may not be available in all Mac Python versions.

Available()
Test availability of the Speech Manager extension (and, on the PowerPC, the Speech Manager shared
library). Return 0 or 1.

Version()
Return the (integer) version number of the Speech Manager.

SpeakString(str)
Utter the string str using the default voice, asynchronously. This aborts any speech that may still be
active from prior SpeakString invocations.
Busy()  
Return the number of speech channels busy, system-wide.

CountVoices()  
Return the number of different voices available.

GetIndVoice(num)  
Return a voice object for voice number num.

14.8.1 voice objects

Voice objects contain the description of a voice. It is currently not yet possible to access the parameters of a voice.

GetGender()  
Return the gender of the voice: 0 for male, 1 for female and −1 for neuter.

NewChannel()  
Return a new speech channel object using this voice.

14.8.2 speech channel objects

A speech channel object allows you to speak strings with slightly more control than SpeakString(), and allows you to use multiple speakers at the same time. Please note that channel pitch and rate are interconnected in some way, so that to make your Macintosh sing you will have to adjust both.

SpeakText(str)  
Start uttering the given string.

Stop()  
Stop babbling.

GetPitch()  
Return the current pitch of the channel, as a floating-point number.

SetPitch(pitch)  
Set the pitch of the channel.

GetRate()  
Get the speech rate (utterances per minute) of the channel as a floating point number.

SetRate(rate)  
Set the speech rate of the channel.

Chapter 15

Standard Windowing Interface

The modules in this chapter are available only on those systems where the STDWIN library is available. STDWIN runs on UNIX under X11 and on the Macintosh. See CWI report CS-R8817.

Warning: Using STDWIN is not recommended for new applications. It has never been ported to Microsoft Windows or Windows NT, and for X11 or the Macintosh it lacks important functionality — in particular, it has no tools for the construction of dialogs. For most platforms, alternative, native solutions exist (though none are currently documented in this manual). Thimber for UNIX under X11, native X with Motif or Athena widgets for UNIX under X11, Win32 for Windows and Windows NT, and a collection of native toolkit interfaces for the Macintosh.

15.1 Built-in Module stdwin

This module defines several new object types and functions that provide access to the functionality of STDWIN. On Unix running X11, it can only be used if the DISPLAY environment variable is set or an explicit ‘-display displayname’ argument is passed to the Python interpreter.

Functions have names that usually resemble their C STDWIN counterparts with the initial ‘w’ dropped. Points are represented by pairs of integers; rectangles by pairs of points. For a complete description of STDWIN please refer to the documentation of STDWIN for C programmers (aforementioned CWI report).

15.1.1 Functions Defined in Module stdwin

The following functions are defined in the stdwin module:

open(title)  
Open a new window whose initial title is given by the string argument. Return a window object; window object methods are described below.\footnote{The Python version of STDWIN does not support draw procedures; all drawing requests are reported as draw events.}

getevent()  
Wait for and return the next event. An event is returned as a triple: the first element is the event type, a small integer; the second element is the window object to which the event applies, or None if it applies to no window in particular; the third element is type-dependent. Names for event types and command codes are defined in the standard module stdwinevent.
pollevent()
    Return the next event, if one is immediately available. If no event is available, return ().

getactive()
    Return the window that is currently active, or None if no window is currently active. (This can be
    emulated by monitoring WE_ACTIVATE and WE_DEACTIVATE events.)

listfontnames(pattern)
    Return the list of font names in the system that match the pattern (a string). The pattern should normally
    be ‘*’, returns all available fonts. If the underlying window system is X11, other patterns follow the
    standard X11 font selection syntax (as used e.g. in resource definitions), i.e. the wildcard character ‘*’
    matches any sequence of characters (including none) and ‘?’ matches any single character. On the
    Macintosh this function currently returns an empty list.

setdefscrollbars(hflag, vflag)
    Set the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll
    bars.

setdefwinpos(h, v)
    Set the default window position for windows opened subsequently.

setdefwinsize(width, height)
    Set the default window size for windows opened subsequently.

setdefscrollbars()
    Return the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll
    bars.

setdefwinpos()
    Return the default window position for windows opened subsequently.

setdefwinsize()
    Return the default window size for windows opened subsequently.

getscrsize()
    Return the screen size in pixels.

getscrmm()
    Return the screen size in millimeters.

fetchcolor(colorname)
    Return the pixel value corresponding to the given color name. Return the default foreground color for
    unknown color names. Hint: the following code tests whether you are on a machine that supports more
    than two colors:
    
    if stdwin.fetchcolor(‘black’) <> \n        stdwin.fetchcolor(‘red’) <> \n        stdwin.fetchcolor(‘white’):
        print ‘color machine’
    else:
        print ‘monochrome machine’

setfgcolor(pixel)
    Set the default foreground color. This will become the default foreground color of windows opened
    subsequently, including dialogs.

setbgcolor(pixel)
    Set the default background color. This will become the default background color of windows opened
    subsequently, including dialogs.

getfgcolor()
    Return the pixel value of the current default foreground color.

getbgcolor()
    Return the pixel value of the current default background color.

setfont(fontname)
    Set the current default font. This will become the default font for windows opened subsequently, and is
    also used by the text measuring functions textwidth, textheight, lineheight and baseline below. This accepts two more optional parameters, size and style. Size is the font size (in ‘points’). Style
    is a single character specifying the style, as follows: ‘b’ = bold, ‘i’ = italic, ‘o’ = bold + italic, ‘u’ = underline; default style is roman. Size and style are ignored under X11 but used on the Macintosh.
    (Sorry for all this complexity — a more uniform interface is being designed.)

menucreate(title)
    Create a menu object referring to a global menu (a menu that appears in all windows). Methods of
    menu objects are described below. Warning: the menu only appears in a window as long as the object returned by
    this call exists.

newbitmap(width, height)
    Create a new bitmap object of the given dimensions. Methods of bitmap objects are described below. Not available on the Macintosh.

fleep()
    Cause a beep or bell (or perhaps a ‘visual bell’ or flash, hence the name).

message(string)
    Display a dialog box containing the string. The user must click OK before the function returns.

askyno(prompt, default)
    Display a dialog that prompts the user to answer a question with yes or no. Return 0 for no, 1 for yes. If
    the user hits the Return key, the default (which must be 0 or 1) is returned. If the user cancels the dialog,
    the KeyboardInterrupt exception is raised.

askstr(prompt, default)
    Display a dialog that prompts the user for a string. If the user hits the Return key, the default string is
    returned. If the user cancels the dialog, the KeyboardInterrupt exception is raised.

askfile(prompt, default, new)
    Ask the user to specify a filename. If new is zero it must be an existing file; otherwise, it must be a new
    file. If the user cancels the dialog, the KeyboardInterrupt exception is raised.

setcutbuffer(i, string)
    Store the string in the system’s cut buffer number i, where it can be found (for pasting) by other
    applications. On X11, there are 8 cut buffers (numbered 0..7). Cut buffer number 0 is the ‘clipboard’ on
    the Macintosh.

getcutbuffer(i)
    Return the contents of the system’s cut buffer number i.

rotatecutbuffers(n)
    On X11, rotate the 8 cut buffers by n. Ignored on the Macintosh.

getselection(i)
    Return X11 selection number i. Selections are not cut buffers. Selection numbers are defined in module
    stdwindows. Selection WS_PRIMARY is the primary selection (used by xterm, for instance); selection WS_SECONDARY is the secondary selection; selection WS_CLIPBOARD is the clipboard
    selection (used by xclipboard). On the Macintosh, this always returns an empty string.
resetselection(i)
Reset selection number i, if this process owns it. (See window method setselection().)

baseline()
Return the baseline of the current font (defined by STD_WIN as the vertical distance between the baseline
and the top of the characters).

lineheight()
Return the total line height of the current font.

textbreak(str, width)
Return the number of characters of the string that fit into a space of width bits wide when drawn in the
current font.

textwidth(str)
Return the width in bits of the string when drawn in the current font.

connectionnumber()

fileno()
(X11 under UNIX only) Return the “connection number” used by the underlying X11 imple-
mentation. (This is normally the file number of the socket.) Both functions return the same value; connectionnumber() is named after the corresponding function in X11 and STD_WIN,
while fileno() makes it possible to use the stdwin module as a “file” object parameter to
select.select(). Note that if select() implies that input is possible on stdwin, this does
not guarantee that an event is ready — it may be some internal communication going on between the X
server and the client library. Thus, you should call stdwin.pollevent() until it returns None to
check for events if you don’t want your program to block. Because of internal buffering in X11, it is
also possible that stdwin.pollevent() returns an event while select() does not find stdwin
to be ready; so you should read any pending events with stdwin.pollevent() until it returns None
before entering a blocking select() call.

15.1.2 Window Objects
Window objects are created by stdwin.open(). They are closed by their close() method or when they
are garbage-collected. Window objects have the following methods:

begindrawing()
Return a drawing object, whose methods (described below) allow drawing in the window.

change(rect)
Invalidates the given rectangle; this may cause a draw event.

gettitle()
Returns the window’s title string.

getdocsize()
Return a pair of integers giving the size of the document as set by setdocsize().

getorign()
Return a pair of integers giving the origin of the window with respect to the document.

gettitle()
Return the window’s title string.

getwinsize()
Return a pair of integers giving the size of the window.

getwinpos()
Return a pair of integers giving the position of the window’s upper left corner (relative to the upper left
corner of the screen).

menucreate(title)
Create a menu object referring to a local menu (a menu that appears only in this window). Methods of
menu objects are described below. Warning: the menu only appears as long as the object returned by
this call exists.

scroll(rect, point)
Scroll the given rectangle by the vector given by the point.

setdocsize(point)
Set the size of the drawing document.

setorign(point)
Move the origin of the window (its upper left corner) to the given point in the document.

setselection(i, str)
Attempt to set X11 selection number i to the string str. (See stdwin method getselection() for the
meaning of i.) Return true if it succeeds. If succeeds, the window “owns” the selection until (a) another
application takes ownership of the selection; or (b) the window is deleted; or (c) the application clears
ownership by calling stdwin.resetselection(i). When another application takes ownership of
the selection, a WE_LOST_SEL event is received for no particular window and with the selection number
as detail. Ignored on the Macintosh.

settimer(dsecs)
Schedule a timer event for the window in dsecs/10 seconds.

settitle(title)
Set the window’s title string.

setwinpos(h, v)
Set the (h,v) position of the window’s upper left corner (relative to the upper left corner of the screen).

setwinsize(width, height)
Set the window’s size.

show(rect)
Try to ensure that the given rectangle of the document is visible in the window.

textcreate(rect)
Create a text-edit object in the document at the given rectangle. Methods of text-edit objects are described
below.

setactive()
Attempt to make this window the active window. If successful, this will generate a WE активЕ activate
event (and a WE_DEACTIVATE event in case another window in this application became inactive).

close()
Discard the window object. It should not be used again.

15.1.3 Drawing Objects
Drawing objects are created exclusively by the window method begindrawing(). Only one drawing object
can exist at any given time; the drawing object must be deleted to finish drawing. No drawing object may exist
when stdwin.getevent() is called. Drawing objects have the following methods:

- **box**(rect)
  - Draw a box just inside a rectangle.

- **circle**(center, radius)
  - Draw a circle with given center point and radius.

- **elarc**(center, (rh, rv), (al, a2))
  - Draw an elliptical arc with given center point. (rh, rv) gives the half sizes of the horizontal and vertical radii. (al, a2) gives the angles (in degrees) of the begin and end points. 0 degrees is at 3 o'clock, 90 degrees is at 12 o'clock.

- **erase**(rect)
  - Erase a rectangle.

- **fillcircle**(center, radius)
  - Draw a filled circle with given center point and radius.

- **fillelarc**(center, (rh, rv), (al, a2))
  - Draw a filled elliptical arc; arguments as for elarc.

- **fillpoly**(points)
  - Draw a filled polygon given by a list (or tuple) of points.

- **invert**(rect)
  - Invert a rectangle.

- **line**(p1, p2)
  - Draw a line from point p1 to p2.

- **paint**(rect)
  - Paint a rectangle.

- **poly**(points)
  - Draw the lines connecting the given list (or tuple) of points.

- **shade**(rect, percent)
  - Fill a rectangle with a shading pattern that is about percent percent filled.

- **text**(p, str)
  - Draw a string starting at point p (the point specifies the top left coordinate of the string).

- **xorcircle**(center, radius)

- **xorelarc**(center, (rh, rv), (al, a2))

- **xorline**(p1, p2)

- **xorpoly**(points)
  - Draw a circle, an elliptical arc, a line or a polygon, respectively, in XOR mode.

- **setfgcolor()**, **setbgcolor()**, **getfgcolor()**, **getbgcolor()**
  - These functions are similar to the corresponding functions described above for the stdwin module, but affect or return the colors currently used for drawing instead of the global default colors. When a drawing object is created, its colors are set to the window’s default colors, which are in turn initialized from the global default colors when the window is created.

- **setfont()**, **baseline()**, **lineheight()**

15.1.4 Menu Objects

A menu object represents a menu. The menu is destroyed when the menu object is deleted. The following methods are defined:

- **additem**(text, shortcut)
  - Add a menu item with given text. The shortcut must be a string of length 1, or omitted (to specify no shortcut).

- **setitem**(i, text)
  - Set the text of item number i.

- **enable**(i, flag)
  - Enable or disable item i.

- **check**(i, flag)
  - Set or clear the check mark for item i.

- **close()**
  - Discard the menu object. It should not be used again.

15.1.5 Bitmap Objects

A bitmap represents a rectangular array of bits. The top left bit has coordinate (0, 0). A bitmap can be drawn with the bitmap method of a drawing object. Bitmaps are currently not available on the Macintosh.

The following methods are defined:

- **bitmap**(points, bitmap, mask)
  - Draw the bitmap with its top left corner at point. If the optional mask argument is present, it should be either the same object as bitmap, to draw only those bits that are set in the bitmap, in the foreground color, or None, to draw all bits (ones are drawn in the foreground color, zeros in the background color). Not available on the Macintosh.

- **cliprect**(rect)
  - Set the “clipping region” to a rectangle. The clipping region limits the effect of all drawing operations, until it is changed again or until the drawing object is closed. When a drawing object is created the clipping region is set to the entire window. When an object to be drawn falls partly outside the clipping region, the set of pixels drawn is the intersection of the clipping region and the set of pixels that would be drawn by the same operation in the absence of a clipping region.

- **noclip()**
  - Reset the clipping region to the entire window.

- **close()**
  - Discard the drawing object. It should not be used again.
15.1.6 Text-edit Objects

A text-edit object represents a text-edit block. For semantics, see the STDWIN documentation for C programmers. The following methods exist:

- `setbit(point, bit)`
  Set the value of the bit indicated by `point` to `bit.`

- `getbit(point)`
  Return the value of the bit indicated by `point`.

- `close()`
  Discard the bitmap object. It should not be used again.

15.1.7 Example

Here is a minimal example of using STDWIN in Python. It creates a window and draws the string “Hello world” in the top left corner of the window. The window will be correctly redrawn when covered and re-exposed. The program quits when the close icon or menu item is requested.

```python
import stdwin
from stdwinevents import *

def main():
    mywin = stdwin.open('Hello')
    
    # while 1:
    (type, win, detail) = stdwin.getevent()
    if type == WE_DRAW:
        draw = win.begindrawing()
        draw.text((0, 0), 'Hello, world')
        del draw
    elif type == WE_CLOSE:
        break

main()
```

15.2 Standard Module stdwinevents

This module defines constants used by STDWIN for event types (WE_ACTIVATE etc.), command codes (WC_LEFT etc.) and selection types (WS_PRIMARY etc.). Read the file for details. Suggested usage is

```python
>>> from stdwinevents import *
>>> ...
```

15.3 Standard Module rect

This module contains useful operations on rectangles. A rectangle is defined as in module stdwin: a pair of points, where a point is a pair of integers. For example, the rectangle

\((10, 20), (90, 80)\)

is a rectangle whose left, top, right and bottom edges are 10, 20, 90 and 80, respectively. Note that the positive vertical axis points down (as in stdwin).

The module defines the following objects:

- `error`
  The exception raised by functions in this module when they detect an error. The exception argument is a string describing the problem in more detail.
empty
The rectangle returned when some operations return an empty result. This makes it possible to quickly check whether a result is empty:

```python
>>> import rect
>>> r1 = (10, 20), (90, 80)
>>> r2 = (0, 0), (10, 20)
>>> r3 = rect.intersect([r1, r2])
>>> if r3 is rect.empty: print 'Empty intersection'
Empty intersection
```

is_empty(r)
Returns true if the given rectangle is empty. A rectangle \((\text{left}, \text{top}), (\text{right}, \text{bottom})\) is empty if \(\text{left} \geq \text{right}\) or \(\text{top} \geq \text{bottom}\).

intersect(list)
Returns the intersection of all rectangles in the list argument. It may also be called with a tuple argument. Raises rect.error if the list is empty. Returns rect.empty if the intersection of the rectangles is empty.

union(list)
Returns the smallest rectangle that contains all non-empty rectangles in the list argument. It may also be called with a tuple argument or with two or more rectangles as arguments. Returns rect.empty if the list is empty or all its rectangles are empty.

pointinrect(point, rect)
Returns true if the point is inside the rectangle. By definition, a point \((h, v)\) is inside a rectangle \((\text{left}, \text{top}), (\text{right}, \text{bottom})\) if \(\text{left} \leq h < \text{right}\) and \(\text{top} \leq v < \text{bottom}\).

inset(rect, (dh, dv))
Returns a rectangle that lies inside the rect argument by \(dh\) pixels horizontally and \(dv\) pixels vertically. If \(dh\) or \(dv\) is negative, the result lies outside rect.

rect2geom(rect)
Converts a rectangle to geometry representation: \((\text{left}, \text{top}), (\text{width}, \text{height})\).

geom2rect(geom)
Converts a rectangle given in geometry representation back to the standard rectangle representation \((\text{left}, \text{top}), (\text{right}, \text{bottom})\).

Chapter 16
SGI IRIX Specific Services

The modules described in this chapter provide interfaces to features that are unique to SGI’s IRIX operating system (versions 4 and 5).

16.1 Built-in Module al

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You’ll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with ‘AL’ prefixed to their name.

Symbolic constants from the C header file ‘<audio.h>’ are defined in the standard module AL, see below.

Warning: the current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size — there is no documented upper limit.)

The module defines the following functions:

openport(name, direction [, config ])
The name and direction arguments are strings. The optional config argument is a configuration object as returned by al.newconfig(). The return value is an port object; methods of port objects are described below.

newconfig()The return value is a new configuration object; methods of configuration objects are described below.

queryparams(device)
The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

getparams(device, list)
The device argument is an integer. The list argument is a list such as returned by queryparams(); it is modified in place (!).

setparams(device, list)
The device argument is an integer. The list argument is a list such as returned by al.queryparams.
16.1.1 Configuration Objects

Configuration objects (returned by \texttt{al.newconfig()}) have the following methods:

- \texttt{getqueuesize()}
  Return the queue size.
- \texttt{setqueuesize(size)}
  Set the queue size.
- \texttt{getwidth()}
  Get the sample width.
- \texttt{setwidth(width)}
  Set the sample width.
- \texttt{getchannels()}
  Get the channel count.
- \texttt{setchannels(channels)}
  Set the channel count.
- \texttt{getsampfmt()}
  Get the sample format.
- \texttt{setsampfmt(sampfmt)}
  Set the sample format.
- \texttt{getfloatmax()}
  Get the maximum value for floating sample formats.
- \texttt{setfloatmax(floatmax)}
  Set the maximum value for floating sample formats.

16.1.2 Port Objects

Port objects (returned by \texttt{al.openport()}) have the following methods:

- \texttt{closeport()}
  Close the port.
- \texttt{getfd()}
  Return the file descriptor as an int.
- \texttt{getfilled()}
  Return the number of filled samples.
- \texttt{getfillable()}
  Return the number of fillable samples.
- \texttt{readamps(samples)}
  Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes.
- \texttt{writamps(samples)}
  Write samples into the queue, blocking if necessary. The samples are encoded as described for the readamps return value.
- \texttt{getfillpoint()}
  Return the 'fill point'.
- \texttt{setfillpoint(fillpoint)}
  Set the 'fill point'.
- \texttt{getconfig()}
  Return a configuration object containing the current configuration of the port.
- \texttt{setconfig(config)}
  Set the configuration from the argument, a configuration object.
- \texttt{getstatus(lst)}
  Get status information on last error.

16.2 Standard Module AL

This module defines symbolic constants needed to use the built-in module \texttt{al} (see above); they are equivalent to those defined in the C header file `<audio.h>` except that the name prefix `AL_` is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import al
from AL import *
```

16.3 Built-in Module cd

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with \texttt{cd.open()} and creates a parser to parse the data from the CD with \texttt{cd.createparser()}. The object returned by \texttt{cd.open()} can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into tracks or programs (the terms are used interchangeably). Tracks can be subdivided into indices. An audio CD contains a table of contents which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module \texttt{cd} defines the following functions and constants:

- \texttt{createparser()}
  Create and return an opaque parser object. The methods of the parser object are described below.
- \texttt{msftoframe(min, sec, frame)}
  Converts a (minutes, seconds, frames) tuple representing time in absolute time code into the corresponding CD frame number.
- \texttt{open([device [mode]])}
  Open the CD-ROM device. The return value is an opaque player object; methods of the player object
are described below. The device is the name of the SCSI device file, e.g. /dev/scs0d4l0, or None. If omitted or None, the hardware inventory is consulted to locate a CD-ROM drive. The mode, if not omitted, should be the string 'r'.

The module defines the following variables:

- **error**: Exception raised on various errors.
- **DATASIZE**: The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type `audio`.
- **BLOCKSIZE**: The size of one block of audio data.

The following variables are states as returned by `getstatus`:

- **READY**: The drive is ready for operation loaded with an audio CD.
- **NODISC**: The drive does not have a CD loaded.
- **CDROM**: The drive is loaded with a CD-ROM. Subsequent read or operate operations will return I/O errors.
- **ERROR**: An error occurred while trying to read the disc or its table of contents.
- **PLAYING**: The drive is in CD player mode playing an audio CD through its audio jacks.
- **PAUSED**: The drive is in CD layer mode with play paused.
- **STILL**: The equivalent of PAUSED on older (non-3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

- **audio**: pnum
- **index**: pt ime
- **at ime**: catalog
- **ident**: control

    Integer constants describing the various types of parser callbacks that can be set by the `addcallback()` method of CD parser objects (see below).

Player objects (returned by `cd.open()`) have the following methods:

- **allowremoval()**: Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.
- **bestreadsize()**: Returns the best value to use for the `num_frames` parameter of the `readdata` method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.
- **close()**: Frees the resources associated with the player object. After calling close, the methods of the object should no longer be used.

- **eject()**: Ejects the caddy from the CD-ROM drive.

- **getstatus()**: Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values:
  - **state**: The state of the CD-ROM drive.
  - **track**: The current track number.
  - **time**: The current time in minutes, seconds, and frames.
  - **first**: The first track number.
  - **last**: The last track number.
  - **scsi_audio**: A string representing the SCSI audio device.

Exceptions raised on various errors.

- **gettrackinfo(track)**: Returns information about the specified track. The returned information is a tuple consisting of two elements, the start time of the track and the duration of the track.

- **msftoblock(min, sec, frame)**: Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logical block number for the given CD-ROM drive. You should use `cd.msftoblock()` rather than `msftoblock()` for comparing times. The logical block number differs from the frame number by an offset required by certain CD-ROM drives.

- **play(start, play)**: Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the CD-ROM drive's headphone and audio jacks (if fitted). Play stops at the end of the disc. `start` is the number of the track at which to start playing the CD. If `play` is 0, the CD will be set to an initial paused state. The method `togglepause()` can then be used to commence play.

- **playabs(min, sec, frame, play)**: Like `play()`, except that the start is given in minutes, seconds, frames instead of a track number.

- **playtrack(start, play)**: Like `play()`, except that playing stops at the end of the track.

- **playtrackabs(track, min, sec, frame, play)**: Like `play()`, except that playing begins at the specified absolute time and ends at the end of the specified track.

- **preventremoval()**: Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

- **readdata(num_frames)**: Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the `parseframe` method of the parser object.

- **seek(min, sec, frame)**: Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to an absolute time code location specified in minutes, seconds, and frames. The return value is the logical block number to which the pointer has been set.

- **seekblock(block)**: Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

- **seektrack(track)**: Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM.
The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

stop()

Stops the current playing operation.

togglepause()

Pauses the CD if it is playing, and makes it play if it is paused.

Parser objects (returned by cd.createparser()) have the following methods:

addcallback(type, func, arg)

Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the cd module level (see above). The callback is called as follows: func(arg, type, data), where arg is the user supplied argument, type is the particular type of callback, and data is the data returned for this type of callback. The type of the data depends on the type of callback as follows:

cd.audio: The argument is a string which can be passed unmodified to al.writesamps() .
cd.pnum: The argument is an integer giving the program (track) number.
cd.index: The argument is an integer giving the index number.
cd.ptime: The argument is a tuple consisting of the program time in minutes, seconds, and frames.
cd.atime: The argument is a tuple consisting of the absolute time in minutes, seconds, and frames.
cd.catalog: The argument is a string of 13 characters, giving the catalog number of the CD.
cd.ident: The argument is a string of 12 characters, giving the BRC identification number of the CD.
cd.control: The argument is an integer giving the control bits from the CD subcode data.

deleteparser()

Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

parseframe(frame)

Parses one or more frames of digital audio data from a CD such as returned by readdata. It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then parseframe executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

removecallback(type)

Removes the callback for the given type.

resetparser()

Resets the fields of the parser used for tracking subcodes to an initial state. resetparser should be called after the disc has been changed.

16.4 Built-in Module fl

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous ftp from host `ftp.cs.ruu.nl`. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial 'fl_' from their name. Constants used by the library are defined in module fl, described below.

The creation of objects is a little different in Python than in C: instead of the `current form` maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions fl_addto_form and fl_and_form, and the equivalent of fl_bgn_form is called fl_make_form.

Watch out for the somewhat confusing terminology: FORMS uses the `object` for the buttons, sliders etc. that you can place in a form. In Python, `object` means any value. The Python interface to FORMS introduces two new Python object types: form objects (representing an entire form) and FORMS objects (representing one button, slider etc.). Hopefully this isn’t too confusing.

There are no ‘free objects’ in the Python interface to FORMS, nor is there an easy way to add object classes written in Python. The FORMS interface to GL event handling is available, though, so you can mix FORMS with pure GL windows.

Phaser note: importing fl implies a call to the GL function foreground() and to the FORMS routine fl_init().

16.4.1 Functions Defined in Module fl

Module fl defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

make_form(type, width, height)

Create a form with given type, width and height. This returns a form object, whose methods are described below.

do_forms()

The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value FL.EVT.

call_event_callback(function)

Set the event callback function.

set_event_callback(event)

Set the graphics modes.

get_rgbmode()

Return the current rgb mode. This is the value of the C global variable f1_rgbmode.

show_message(str1, str2, str3)

Show a dialog box with a three-line message and an OK button.

show_question(str1, str2, str3)

Show a dialog box with a three-line message and YES and NO buttons. It returns 1 if the user pressed YES, 0 if NO.

show_choice(str1, str2, str3, but1, but2, but3)

Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (1, 2 or 3).

show_input(prompt, default)

Show a dialog box with a one-line prompt message and text field in which the user can enter a string.

The second argument is the default input string. It returns the string value as edited by the user.
show_file_selector(message, directory, pattern, default)
Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or None if the user presses Cancel.

get_directory()”
get_pattern()”
get_filename()”
These functions return the directory, pattern and filename (the tail part only) selected by the user in the last show_file_selector call.

gdevice(dev)
unqdevice(dev)
 These functions are the FORMS interfaces to the corresponding GL functions. Use these if you want to handle some GL events yourself when using fl.do_events. When a GL event is detected that FORMS cannot handle, fl.do_events() returns the special value FL.EVENT and you should call fl.qread() to read the event from the queue. Don’t use the equivalent GL functions!

color()”
mapcolor()”
getmcolor()”
See the description in the FORMS documentation of fl_color, fl_mapcolor and fl_getmcolor.

16.4.2 Form Objects
Form objects (returned by fl.make_form() above) have the following methods. Each method corresponds to a C function whose name is prefixed with ‘fl_’ ; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the ‘add_’ functions return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

show_form(placement, bordertype, name)
Show the form.

hide_form()”
Redraw the form.

set_form_position(x, y)
Set the form’s position.

freeze_form()”
Freeze the form.

unfreeze_form()”
Unfreeze the form.

activate_form()”
Activate the form.

deactivate_form()”
Deactivate the form.

bgn_group()”
Begin a new group of objects; return a group object.

del_group()”
End the current group of objects.

find_first()”
Find the first object in the form.

find_last()”
Find the last object in the form.

add_box(type, x, y, w, h, name)
Add a box object to the form. No extra methods.

add_text(type, x, y, w, h, name)
Add a text object to the form. No extra methods.

add_lightbutton(type, x, y, w, h, name)
Add a lightbutton object to the form.

add_roundbutton(type, x, y, w, h, name)
Add a roundbutton object to the form.

add_dial(type, x, y, w, h, name)
Add a dial object to the form.

add_slider(type, x, y, w, h, name)
Add a slider object to the form.

add_valslider(type, x, y, w, h, name)
Add a valslider object to the form.

add_clock(type, x, y, w, h, name)
Add a clock object to the form.

add_positioner(type, x, y, w, h, name)
Add a positioner object to the form.

activate_form()”
Activate the form.

deactivate_form()”
Deactivate the form.

bgn_group()”
Begin a new group of objects; return a group object.

del_group()”
End the current group of objects.

find_first()”
Find the first object in the form.

find_last()”
Find the last object in the form.

add_box(type, x, y, w, h, name)
Add a box object to the form. No extra methods.

add_text(type, x, y, w, h, name)
Add a text object to the form. No extra methods.

add_lightbutton(type, x, y, w, h, name)
Add a lightbutton object to the form.

add_roundbutton(type, x, y, w, h, name)
Add a roundbutton object to the form.

add_dial(type, x, y, w, h, name)
Add a dial object to the form.

add_slider(type, x, y, w, h, name)
Add a slider object to the form.

add_valslider(type, x, y, w, h, name)
Add a valslider object to the form.

add_clock(type, x, y, w, h, name)
Add a clock object to the form.

add_positioner(type, x, y, w, h, name)
Add a positioner object to the form.

activate_form()”
Activate the form.

deactivate_form()”
Deactivate the form.

bgn_group()”
Begin a new group of objects; return a group object.

del_group()”
End the current group of objects.

find_first()”
Find the first object in the form.

find_last()”
Find the last object in the form.

add_box(type, x, y, w, h, name)
Add a box object to the form. No extra methods.

add_text(type, x, y, w, h, name)
Add a text object to the form. No extra methods.

add_lightbutton(type, x, y, w, h, name)
Add a lightbutton object to the form.

add_roundbutton(type, x, y, w, h, name)
Add a roundbutton object to the form.

add_dial(type, x, y, w, h, name)
Add a dial object to the form.

add_slider(type, x, y, w, h, name)
Add a slider object to the form.

add_valslider(type, x, y, w, h, name)
Add a valslider object to the form.

add_clock(type, x, y, w, h, name)
Add a clock object to the form.

add_positioner(type, x, y, w, h, name)
Add a positioner object to the form.
add_counter(type, x, y, w, h, name)
    Add a counter object to the form.
    Methods: set_counter_value, get_counter_value, set_counter_bounds,
              set_counter_step, set_counter_precision, set_counter_return.

add_input(type, x, y, w, h, name)
    Add an input object to the form.
    Methods: set_input, get_input, set_input_color, set_input_return.

add_menu(type, x, y, w, h, name)
    Add a menu object to the form.
    Methods: set_menu, get_menu, addto_menu.

add_choice(type, x, y, w, h, name)
    Add a choice object to the form.
    Methods: set_choice, get_choice, clear_choice, addto_choice,
              delete_choice, get_choice_text, set_choice_fontsize,
              set_choice_fontstyle.

add_browser(type, x, y, w, h, name)
    Add a browser object to the form.
    Methods: set_browser_topline, clear_browser, add_browser_line,
              addto_browser, insert_browser_line, delete_browser_line,
              replace_browser_line, get_browser_line, load_browser,
              get_browser_maxline, select_browser_line, deselect_browser_line,
              deselected_browser_line, get_browser,
              set_browser_fontsize, set_browser_fontstyle, set_browser_specialkey.

add_timer(type, x, y, w, h, name)
    Add a timer object to the form.
    Methods: set_timer, get_timer.

Form objects have the following data attributes; see the FORMS documentation:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>window</td>
<td>int (read-only)</td>
<td>GL window id</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>form width</td>
</tr>
<tr>
<td>h</td>
<td>float</td>
<td>form height</td>
</tr>
<tr>
<td>x</td>
<td>float</td>
<td>form x origin</td>
</tr>
<tr>
<td>y</td>
<td>float</td>
<td>form y origin</td>
</tr>
<tr>
<td>deactivated</td>
<td>int</td>
<td>nonzero if form is deactivated</td>
</tr>
<tr>
<td>visible</td>
<td>int</td>
<td>nonzero if form is visible</td>
</tr>
<tr>
<td>frozen</td>
<td>int</td>
<td>nonzero if form is frozen</td>
</tr>
<tr>
<td>doublebuf</td>
<td>int</td>
<td>nonzero if double buffering on</td>
</tr>
</tbody>
</table>

16.4.3 FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

set_call_back(function, argument)
    Set the object's callback function and argument. When the object needs interaction, the callback function
    will be called with two arguments: the object, and the callback argument. (FORMS objects without
    a callback function are returned by fl.do_forms() or fl.check_forms() when they need
    interaction.) Call this method without arguments to remove the callback function.

16.5 Standard Module FL

This module defines symbolic constants needed to use the built-in module fl (see above); they are equivalent
to those defined in the C header file `<forms.h>` except that the name prefix 'FL_' is omitted. Read the
module source for a complete list of the defined names. Suggested use:

```python
import fl
from FL import *
```
16.6 Standard Module flp

This module defines functions that can read form definitions created by the 'form designer' (fdesign) program that comes with the FORMS library (see module f1 above).

For now, see the file 'flp.doc' in the Python library source directory for a description.

XXX A complete description should be inserted here!

16.7 Built-in Module fm

This module provides access to the IRIS Font Manager library. It is available only on Silicon Graphics machines.

See also: 4Sight User's Guide, Section 1, Chapter 5: Using the IRIS Font Manager.

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

- **init()**
  - Initialization function. Calls fminit(). It is normally not necessary to call this function, since it is called automatically the first time the fm module is imported.

- **findfont(fontname)**
  - Return a font handle object. Calls fmfindfont(fontname).

- **enumerate()**
  - Returns a list of available font names. This is an interface to fmenumerate().

- **prstr(string)**
  - Render a string using the current font (see the setfont() font handle method below). Calls fmprstr(string).

- **setpath(string)**
  - Sets the font search path. Calls fmsetpath(string). (XXX Does not work?!!!)

- **fontpath()**
  - Returns the current font search path.

Font handle objects support the following operations:

- **scalefont(factor)**
  - Returns a handle for a scaled version of this font. Calls fmscalefont(fh, factor).

- **setfont()**
  - Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted.

  Calls fmsetfont(fh).

- **getfontname()**
  - Returns this font's name. Calls fmgetfontname(fh).

- **getcomment()**
  - Returns the comment string associated with this font. Raises an exception if there is none. Calls fmgetcomment(fh).

- **getfontinfo()**
  - Returns a tuple giving some pertinent data about this font. This is an interface to fmgetfontinfo(). The returned tuple contains the following numbers: (printermatched, fixed_width, xorig, yorig, xsize, ysize, height, nglyphs).

16.8 Built-in Module gl

This module provides access to the Silicon Graphics Graphics Library. It is available only on Silicon Graphics machines.

**Warning:** Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

- All (short, long, unsigned) int values are represented by Python integers.
- All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.
- All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.
- All string and character arguments are represented by Python strings, for instance, winopen('Hi There!') and rotate(900, 'z').
- All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call

  ```c
  lndef(deftype, index, np, props)
  ```

  is translated to Python as

  ```python
  lndef(deftype, index, props)
  ```

- Output arguments are omitted from the argument list; they are transmitted as function return values instead. If more than one value must be returned, the return value is a tuple. If the C function has both a regular return value (that is not omitted because of the previous rule) and an output argument, the return value comes first in the tuple. Examples: the C call

  ```c
  getmcolor(i, &red, &green, &blue)
  ```

  is translated to Python as

  ```python
  red, green, blue = getmcolor(i)
  ```

The following functions are non-standard or have special argument conventions:
varray (argument)

Equivalent to but faster than a number of v3d() calls. The argument is a list (or tuple) of points. Each point must be a tuple of coordinates (x, y, z) or (x, y). The points may be 2- or 3-dimensional but must all have the same dimension. Float and int values may be mixed however. The points are always converted to 3D double precision points by assuming z = 0.0 if necessary (as indicated in the man page), and for each point v3d() is called.

nvarray()

Equivalent to but faster than a number of n3f and v3f calls. The argument is an array (list or tuple) of pairs of normals and points. Each pair is a tuple of a point and a normal for that point. Each point or normal must be a tuple of coordinates (x, y, z). Three coordinates must be given. Float and int values may be mixed. For each pair, n3f() is called for the normal, and then v3f() is called for the point.

vnarray()

Similar to nvarray() but the pairs have the point first and the normal second.

nurbsurface(s_k, t_k, cl, s_ord, t_ord, type)

Defines a nurbs surface. The dimensions of cl[][] are computed as follows: len(s_k) - s_ord, len(t_k) - t_ord.

nurbscurve (knots, ctlpoints, order, type)

Defines a nurbs curve. The length of ctlpoints is len (knots) - order.

pwlcurve (points, type)

Defines a piecewise-linear curve. points is a list of points. type must be N_ST.

pick (n)

The only argument to these functions specifies the desired size of the pick or select buffer.

endpick ()

These functions have no arguments. They return a list of integers representing the used part of the pick/select buffer. No method is provided to detect buffer overrun.

Here is a tiny but complete example GL program in Python:

```python
import gl, GL, time

def main():
    gl.foreground()
    gl.prefposition(500, 900, 500, 900)
    w = gl.winopen("CrissCross")
    gl.ortho2(0.0, 400.0, 0.0, 400.0)
    gl.color(GL.WHITE)
    gl.clear()
    gl.color(GL.RED)
    gl.bgnline()
    gl.v2f(0.0, 0.0)
    gl.v2f(400.0, 400.0)
    gl.endline()
    gl.bgnline()
    gl.v2f(400.0, 0.0)
    gl.v2f(0.0, 400.0)
    gl.endline()
    time.sleep(5)
    main()
```

16.9 Standard Modules GL and DEVICE

These modules define the constants used by the Silicon Graphics Graphics Library that C programmers find in the header files `<gl/gl.h>` and `<gl/device.h>`. Read the module source files for details.

16.10 Built-in Module imgfile

The imgfile module allows python programs to access SGI imglib image files (also known as `.rgb` files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

error

This exception is raised on all errors, such as unsupported file type, etc.

getsizes (file)

This function returns a tuple (x, y, z) where x and y are the size of the image in pixels and z is the number of bytes per pixel. Only 3 byte RGB pixels and 1 byte greyscale pixels are currently supported.

read (file)

This function reads and decodes the image on the specified file, and returns it as a python string. The string has either 1 byte greyscale pixels or 4 byte RGBA pixels. The bottom left pixel is the first in the string. This format is suitable to pass to gl.lrectwrite, for instance.

readscaled (file, x, y, filter [, blur])

This function is identical to read but it returns an image that is scaled to the given x and y sizes. If the filter and blur parameters are omitted scaling is done by simply dropping or duplicating pixels, so the result will be less than perfect, especially for computer generated images.
Alternatively, you can specify a filter to use to smoothen the image after scaling. The filter forms supported are 'impulse', 'box', 'triangle', 'quadratic' and 'gaussian'. If a filter is specified blur is an optional parameter specifying the blurriness of the filter. It defaults to 1.0. readscaled makes no attempt to keep the aspect ratio correct, so that is the users' responsibility.

ttob(flag)
This function sets a global flag which defines whether the scan lines of the image are read or written from bottom to top (flag is zero, compatible with SGI GL) or from top to bottom (flag is one, compatible with X). The default is zero.

write(file, data, x, y, z)
This function writes the RGB or greyscale data in data to file file. x and y give the size of the image. z is 1 for 1 byte greyscale images or 3 for RGB images (which are stored as 4 byte values of which only the lower three bytes are used). These are the formats returned by gl.lrectread.

Chapter 17
SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to the SunOS operating system (versions 4 and 5; the latter is also known as Solaris version 2).

17.1 Built-in Module sunaudiodev

This module allows you to access the sun audio interface. The sun audio hardware is capable of recording and playing back audio data in U-LAW format with a sample rate of 8K per second. A full description can be gotten with 'man audio'.

The module defines the following variables and functions:

error
This exception is raised on all errors. The argument is a string describing what went wrong.

open(mode)
This function opens the audio device and returns a sun audio device object. This object can then be used to do I/O on. The mode parameter is one of 'r' for record-only access, 'w' for play-only access, 'rw' for both and 'control' for access to the control device. Since only one process is allowed to have the recorder or player open at the same time it is a good idea to open the device only for the activity needed. See the audio manpage for details.

17.1.1 Audio Device Objects

The audio device objects are returned by open define the following methods (except control objects which only provide getinfo, setinfo and drain):

close()
This method explicitly closes the device. It is useful in situations where deleting the object does not immediately close it since there are other references to it. A closed device should not be used again.

drain()
This method waits until all pending output is processed and then returns. Calling this method is often not necessary: destroying the object will automatically close the audio device and this will do an implicit drain.

flush()
This method discards all pending output. It can be used avoid the slow response to a user's stop request.
getinfo()

This method retrieves status information like input and output volume, etc. and returns it in the form of an audio status object. This object has no methods but it contains a member of attributes describing the current device status. The names and meanings of the attributes are described in '/usr/include/sun/audioio.h' and in the audiomappage. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the play structure have '_o_' prepended to their name and members of the record structure have '_i_' prepended. So, the C member play.sample_rate is accessed as o_sample_rate, record.gain as i_gain and monitor_gain plainly as monitor_gain.

obufcount ()

This method returns the number of samples that are buffered on the recording side, i.e. the program will not block on a read call of so many samples.

obufcount ()

This method returns the number of samples buffered on the playback side. Unfortunately, this number cannot be used to determine a number of samples that can be written without blocking since the kernel output queue length seems to be variable.

read (size)

This method reads size samples from the audio input and returns them as a Python string. The function blocks until enough data is available.

setinfo (status)

This method sets the audio device status parameters. The status parameter is an device status object as returned by getinfo and possibly modified by the program.

write (samples)

Write is passed a Python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

There is a companion module, 'sunaudioio', which defines useful symbolic constants like MIN_GAIN, MAX_GAIN, SPEAKER, etc. The names of the constants are the same names as used in the C include file '<sun/audioio.h>', with the leading string 'AUDIO_' stripped.

Useability of the control device is limited at the moment, since there is no way to use the "wait for something to happen" feature the device provides.

Index

```
--
operator, 4
__builtin___ (built-in module), 34
__dict___ (pickle protocol), 27
__getinitargs__ (copy protocol), 30
__getstate__ (pickle protocol), 27
__init__ (pickle protocol), 30
__init___ (pickle protocol), 27
__main__ (built-in module), 34
__setstate__ (copy protocol), 30
__setstate___ (pickle protocol), 27
_exit (in module posix), 60

Unix

file control, 68
I/O control, 68

ABC

language, 4

Abort (TCP stream method), 129
Aborted (connection object method), 124
abort (FTP object method), 97
abs (built-in function), 14
accept (connection object method), 123
accept (socket method), 54
acquire (lock method), 58
activate_form (form object method), 150
ActiveOpen (TCP stream method), 129
ActiveOpen (TCP stream method), 129
add (Stats method), 82
add (in module audio), 110
add_box (form object method), 150
add_browser (form object method), 151
add_button (form object method), 150
add_choice (form object method), 151
add_clock (form object method), 150
add_counter (form object method), 151
add_dial (form object method), 150
add_input (form object method), 151
add_lightbutton (form object method), 150
add_menu (form object method), 151
add_positioner (form object method), 150
add_roundbutton (form object method), 150
add_slider (form object method), 150
add_text (form object method), 150

add_timer (form object method), 151
add_valselider (form object method), 150
addcalliback (CD parser object method), 147
additem (menu method), 138
AddrToName (in module macdnr), 126
AtrrToStr (in module macdnr), 126
add_timer (CD parser object method), 147
addcallback (in module macdnr), 126
addtimer (in module audio), 110
(adpcm32lin (in module audio), 110
AF_INET (in module socket), 53
AF_UNIX (in module socket), 53
allf (aifc object method), 115
allf (aifc object method), 115
AL (standard module), 144

alarm (in module signal), 51
all_errors (in module stdio), 96
allocate_lock (in module thread), 58
allowremoval (CD player object method), 145
altzone (in module time), 47
amtUnackedData (TCP status attribute), 130
amtUnreadData (TCP status attribute), 130
Anchor (in module stdwin), 134
anchors (in module stdwin), 134
anchor (in module stdwin), 134
anchor (in module stdwin), 134
anchor (in module stdwin), 134

and

operator, 3
append (in module array), 44
append (list method), 8
apply (built-in function), 14
arg (in module sys), 21
arithmetic, 4
array (built-in module), 40, 43
array (in module array), 43
arrays, 43
arrow (text-edit method), 139
article (NNTP object method), 101
as_path (in module fs), 128
as_tuple (FS object method), 128
asctime (in module time), 47
askfile (in module stdwin), 134
askstr (in module stdwin), 134
```