Python Library Reference

Guido van Rossum
Dept. CST, CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
E-mail: guido@cwi.nl

26 Januari 1994
Release 1.0.0

Abstract

This document describes the built-in types, exceptions and functions and the standard modules that come with the Python system. It assumes basic knowledge about the Python language. For an informal introduction to the language, see the *Python Tutorial*. The *Python Reference Manual* gives a more formal definition of the language.

Contents

Chapter 1

Introduction

The Python library consists of three parts, with different levels of integration with the interpreter. Closest to the interpreter are built-in types, exceptions and functions. Next are built-in modules, which are written in C and linked statically with the interpreter. Finally there are standard modules that are implemented entirely in Python, but are always available. For efficiency, some standard modules may become built-in modules in future versions of the interpreter.

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, so local and global user-defined names can override built-in names. Built-in types have no names but are created easily by constructing an object of the desired type (e.g., using a literal) and applying the built-in function `type()` to it. They are described together here for easy reference.[1]

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 `` `...` `` 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 false:

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

[1]Some descriptions sorely lack explanations of the exceptions that may be raised — this will be fixed in a future version of this document.

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

2.1.2 Boolean Operations

These are the Boolean operations:

Operation	Result	Notes
x `or` y	if x is false, then y, else x	(1)
x `and` y	if x is false, then x, else y	(1)
`not` x	if x is false, then `1`, else `0`	

Notes:

(1) These only evaluate their second argument if needed for their outcome.

2.1.3 Comparisons

Comparison operations are supported by all objects:

Operation	Meaning	Notes
`<`	strictly less than	
`<=`	less than or equal	
`>`	strictly greater than	
`>=`	greater than or equal	
`==`	equal	
`<>`	not equal	(1)
`!=`	not equal	(1)
`is`	object identity	
`is not`	negated object identity	

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.[2] The functions `int()`, `long()` and `float()` can be used to coerce numbers to a specific type.

All numeric types support the following operations:

Operation	Result	Notes
`abs(`*x*`)`	absolute value of *x*	
`int(`*x*`)`	*x* converted to integer	(1)
`long(`*x*`)`	*x* converted to long integer	(1)
`float(`*x*`)`	*x* converted to floating point	
`-`*x*	*x* negated	
`+`*x*	*x* unchanged	
x `+` *y*	sum of *x* and *y*	
x `-` *y*	difference of *x* and *y*	
x `*` *y*	product of *x* and *y*	
x `/` *y*	quotient of *x* and *y*	(2)
x `%` *y*	remainder of *x* `/` *y*	
`divmod(`*x*`,` *y*`)`	the pair `(`*x* `/` *y*`,` *x* `%` *y*`)`	(3)
`pow(`*x*`,` *y*`)`	*x* to the power *y*	

Notes:

(1) 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.

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

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

[2] As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similar for tuples.

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:

Operation	Result	Notes
`~`x	the bits of x inverted	
x `^` y	bitwise *exclusive or* of x and y	
x `&` y	bitwise *and* of x and y	
x `|` y	bitwise *or* of x and y	
x `<<` n	x shifted left by n bits	
x `>>` n	x shifted right by n bits	

2.1.5 Sequence Types

There are three sequence types: strings, lists and tuples. Strings literals are written in single quotes: `'xyzzy'`. 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, b, c` or `()`. A single item tuple must have a trailing comma, e.g., `(d,)`.

Sequence types support the following operations (s and t are sequences of the same type; n, i and j are integers):

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

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. If i or j is greater than `len(`s`)`, use `len(`s`)`. If i is omitted, use `0`. If j is omitted, use `len(`s`)`. If i is greater than or equal to j, the slice is empty.

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 sprintf format string to be applied to the right argument, and returns the string resulting from this formatting operation.

Unless the format string requires exactly one argument, the right argument should be a tuple of the correct size. 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 may be present but are ignored. 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, huge floating point precisions are truncated; `%f` conversions for huge numbers are replaced by `%g` conversions. All other errors raise exceptions.

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):

Operation	Result	Notes
s`[`*i*`] = `*x*	item i of s is replaced by x	
s`[`*i*`:`*j*`] = `*t*	slice of s from i to j is replaced by t	
`del ` *s*`[`*i*`:`*j*`]`	same as *s*`[`*i*`:`*j*`] = []`	
s`.append(`*x*`)`	same as *s*`[len(`*x*`):len(`*x*`)] = [`*x*`]`	
s`.count(`*x*`)`	return number of i's for which *s*`[`*i*`] == `*x*	
s`.index(`*x*`)`	return smallest i such that *s*`[`*i*`] == `*x*	(1)
s`.insert(`*i*`, `*x*`)`	same as *s*`[`*i*`:`*i*`] = [`*x*`]`	
s`.remove(`*x*`)`	same as `del ` *s*`[`*s*`.index(`*x*`)]`	(1)
s`.reverse()`	reverses the items of s in place	
s`.sort()`	permutes the items of s to satisfy *s*`[`*i*`] <= `*s*`[`*j*`]`, for *i* `<` *j*	(2)

Notes:

(1) Raises an exception when x is not found in s.

(2) The `sort()` method takes an optional argument specifying a comparison function of two arguments (list items) which should return `-1`, `0` or `1` depending on whether the first argument is considered smaller than, equal to, or larger than the second argument. Note that this slows the sorting process down considerably; e.g. to sort an array in reverse order it is much faster to use calls to `sort()` and `reverse()` than to use `sort()` with a comparison function that reverses the ordering of the elements.

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 mapping type, the *dictionary*. A dictionary's keys are almost arbitrary values. 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': 4098, 'sjoerd: 4127}` or `{4098: '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):

Operation	Result	Notes
`len(`*a*`)`	the number of items in a	
a`[`*k*`]`	the item of a with key k	(1)
a`[`*k*`] =` *x*	set *a*`[`*k*`]` to x	
`del` *a*`[`*k*`]`	remove *a*`[`*k*`]` from a	(1)
a`.items()`	a copy of a's list of (key, item) pairs	(2)
a`.keys()`	a copy of a's list of keys	(2)
a`.values()`	a copy of a's list of values	(2)
a`.has_key(`*k*`)`	`1` if a has a key k, else `0`	

Notes:

(1) Raises an exception if k is not in the map.

(2) Keys and values are listed in random order, but at any moment the ordering of the `keys()`, `values()` and `items()` lists is the consistent with each other.

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

A special member of every module is `__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 `__dict__` attribute is not possible (i.e., you can write *m*`.__dict__['a'] = 1`, which defines *m*`.a` to be `1`, but you can't write *m*`.__dict__ = {}`.

Modules are written like this: `<module 'sys'>`.

Classes and Class Instances.

(See 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: *func*`(`*argument-list*`)`.

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: *f*`.func_code` is a function's *code object* (see below) and *f*`.func_globals` is the dictionary used as the function's global name space (this is the same as *m*`.__dict__` where *m* is the module in which the function *f* was defined).

Methods.

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as `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: *m*`.im_self` is the object whose method this is, and *m*`.im_func` is the function implementing the method. Calling *m*`(`*arg-1*`, `*arg-2*`, ..., `*arg-n*`)` is completely equivalent to calling *m*`.im_func(`*m*`.im_self, `*arg-1*`, `*arg-2*`, ..., `*arg-n*`)`.

(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 function `type()`. There are no special operations on types.

Types are written like this: `<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 `None` (a built-in name).

It is written as `None`.

File Objects.

File objects are implemented using C's `stdio` package and can be created with the built-in function `open()` described under Built-in Functions below.

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

Files have the following methods:

`close()`
: Close the file. A closed file cannot be read or written anymore.

`flush()`
: Flush the internal buffer, like `stdio`'s `fflush()`.

`isatty()`
: Return `1` if the file is connected to a tty(-like) device, else `0`.

`read(`*size*`)`
: Read at most *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 (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.

`seek(`*offset*`, `*whence*`)`
: Set the file's current position, like `stdio`'s `fseek()`. The *whence* argument is optional and defaults to `0` (absolute file positioning); other values are `1` (seek relative to the current position) and `2` (seek relative to the file's end). There is no return value.

`tell()`
: Return the file's current position, like `stdio`'s `ftell()`.

`write(`*str*`)`
: Write a string to the file. There is no return value.

Internal Objects.

(See the Python Reference Manual for these.)

2.1.8 Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant:

- *x*.`__dict__` is a dictionary of some sort used to store an object's (writable) attributes;
- *x*.`__methods__` lists the methods of many built-in object types, e.g., `[].__methods__` is `['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort']`;
- *x*.`__members__` lists data attributes;
- *x*.`__class__` is the class to which a class instance belongs;
- *x*.`__bases__` is the tuple of base classes of a class object.

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 attributes 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(`*x*`)`
: 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.g., `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 a `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.

`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:

```
>>> 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 `(math.floor(`*a* `/` *b*`),` *a* `%` *b*`)`.

`eval(`*s*`, `*globals*`, `*locals*`)`

The arguments are a string and two optional dictionaries. The string argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the dictionaries as global and local name space. The string must not contain null bytes or newline characters. The return value is the result of the expression. If the third argument is omitted it defaults to the second. If both dictionaries are omitted, the expression is executed in the environment where `eval` is called. Syntax errors are reported as exceptions. Example:

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

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

Note: dynamic execution of statements is supported by the `exec` statement.

`filter(`*function*`, `*list*`)`
: Construct a list from those elements of *list* for which *function* returns true. If *list* is a string or a tuple, the result also has that type; otherwise it is always a list. If *function* is `None`, the identity function is assumed, i.e. all elements of *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, `getattr(`*x*`, '`*foobar*`')` is equivalent to *x*`.`*foobar*.

`hasattr(`*object*`, `*name*`)`
: 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 `getattr(object, name)` and seeing whether it raises an exception or not.)

`hash(`*object*`)`
: Return the hash value of the object (if it has one). Hash values are 32-bit integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, e.g. 1 and 1.0).

`hex(`*x*`)`
: Convert a number 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 disjunct may have the same id() value.) (Implementation note: this is the address of the object.)

`input(`*prompt*`)`
: Almost equivalent to `eval(raw_input(`*prompt*`))`. As for `raw_input()`, the 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.

`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.

`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).

`oct(`*x*`)`
: Convert a number to an octal string. The result is a valid Python expression.

`open(`*filename*`, `*mode*`)`
: Return a new file object (described earlier under Built-in Types). The string 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, `IOError` is raised.

`ord(`*c*`)`
: Return the ASCII value of a string of one character. E.g., `ord('a')` returns the integer `97`. This is the inverse of `chr()`.

`pow(`*x*`, `*y*`)`
: Return x to the power y. 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)` 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. Example:

```
>>> 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)
[]
>>> range(1, 0)
[]
>>>
```

`raw_input(`*prompt*`)`
: The string argument is optional; if 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:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
'Monty Python\'s Flying Circus'
>>>
```

`reduce(`*function*`, `*list*`, `*initializer*`)`
: Apply the binary *function* to the items of *list* so as to reduce the list to a single value. E.g., `reduce(lambda x, y:  x*y,` *list*`, 1)` returns the product of the elements of *list*. The optional *initializer* can be thought of as being prepended to *list* so as to allow reduction of an empty *list*. The *list* arguments may be any kind of sequence.

`reload(`*module*`)`
: Re-parse and re-initialize an already imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. Note that if a module is syntactically correct but its initialization fails, the first `import` statement for it does not import the name, but does create a (partially initialized) module object; to reload the module you must first `import` it again (this will just make the partially initialized module object available) before you can `reload()` it.

`repr(`*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.

`type(`*object*`)`

Return the type of an *object*. The return value is a type object. There is not much you can do with type objects except compare them to other type objects; e.g., the following checks if a variable is a string:

```
>>> if type(x) == type(''): print 'It is a string'
```

Chapter 3

Built-in Modules

The modules described in this chapter are built into the interpreter and considered part of Python's standard environment: they are always avaialble.[1]

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. If no script name was passed to the Python interpreter, `sys.argv` is empty.

`builtin_module_names`
: A list of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `sys.modules.keys()` only lists the imported modules.)

`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 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

[1] at least in theory — it is possible to specify at build time that one or more of these modules should be excluded, but it would be antisocial to do so.

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 (but 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 cause 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_tracaback`, respectively.

`modules`
: Gives the list of modules that have already been loaded. This can be manipulated to force reloading 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 `'... '`.

`settrace(`*tracefunc*`)`
: Set the system's trace function, which allows you to implement a Python source code debugger in Python. The standard modules `pdb` and `wdb` are such debuggers; the difference is that `wdb` uses windows and needs STDWIN, while `pdb` has a line-oriented interface not unlike dbx. See the file '`pdb.doc`' in the Python library source directory for more documentation (both about `pdb` and `sys.trace`).

`setprofile(`*profilefunc*`)`
: Set the system's profile function, which allows you to implement a Python source code profiler in Python. 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.stdout` and `sys.stderr` needn't be built-in file objects: any object is acceptable as long as it has a `write` method that takes a string argument.

`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 Built-in Module `__builtin__`

This module provides direct access to all 'built-in' identifier of Python; e.g. `__builtin__.open` is the full name for the built-in function `open`.

3.3 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.

3.4 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:

Typecode	Type	Minimal size in bytes
`'c'`	character	1
`'b'`	signed integer	1
`'h'`	signed integer	2
`'i'`	signed integer	2
`'l'`	signed integer	4
`'f'`	floating point	4
`'d'`	floating point	8

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

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 ffrom 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.

`fromlist(`*list*`)`
: Appends 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*`)`
: Appends 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 array of machine values and return the string representation (the same sequence of bytes that would be written to a file by the `tofile()` method.)

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 string is guaranteed to be able to be converted back to an array with the same type and value using reverse quotes (` `` `). Examples:

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

3.5 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`)`, `ldexp(`x`, `y`)`, `log(`x`)`, `log10(`x`)`, `modf(`x`)`, `pow(`x`, `y`)`, `sin(`x`)`, `sinh(`x`)`, `sqrt(`x`)`, `tan(`x`)`, `tanh(`x`)`.

Note that `frexp` and `modf` have a different call/return pattern than their C equivalents: they take a single argument and return a pair of values, rather than returning their second return value through an 'output parameter' (there is no such thing in Python).

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

3.6 Built-in Module `time`

This module provides various time-related functions. It is always available. (On some systems, not all functions may exist; e.g. the "milli" variants can't always be implemented.)

An explanation of some terminology and conventions is in order.

- The "epoch" is the point where the time starts. On January 1st 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 the first element of `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 every 1/50th or 1/100th of a second, and on the Mac, it ticks 60 times a second.

Functions and data items are:

`altzone`
: The offset of the local DST timezone, in seconds west of the 0th meridian, if one is defined. 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.

`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*`)`
: Convert a time expressed in seconds since the epoch to a tuple of 9 integers, in UTC: year (e.g. 1993), month (1-12), day (1-31), hour (0-23), minute (0-59), second (0-59), weekday (0-6, monday is 0), julian day (1-366), dst flag (always zero). Fractions of a second are ignored. Note subtle differences with the C function of this name.

`localtime(`*secs*`)`
: Like `gmtime` but converts to local time. The dst flag is set to 1 when DST applies to the given time.

`millisleep(`*msecs*`)`
: Suspend execution for the given number of milliseconds. (Obsolete, you can now use use `sleep` with a floating point argument.)

`millitimer()`
: Return the number of milliseconds of real time elapsed since some point in the past that is fixed per execution of the python interpreter (but may change in each following run). The return value may be negative, and it may wrap around.

`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. An alternative for measuring precise intervals is `millitimer`.

`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.

3.7 Built-in Module `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.

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. *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 '`\section{...}`' headers from a document, you can use this pattern: `'\\\\section{\(.*\)}'`.

The module defines these functions, and an exception:

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

`search(`*pattern*`, `*string*`)`
: 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!).

`compile(`*pattern*`, `*translate*`)`
: Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match` and `search` methods, described below. The optional *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 `i`-th element of the string gives the translation for the character with ASCII code `i`.

  The sequence

```
prog = regex.compile(pat)
result = prog.match(str)
```

  is equivalent to

```
result = regex.match(pat, str)
```

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

`set_syntax(`*flags*`)`
: Set the syntax to be used by future calls to `compile`, `match` and `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 `regex_syntax`; read the file '`regex_syntax.py`' for more information.

`symcomp(`*pattern*`, `*translate*`)`
: This is like `compile`, but supports symbolic group names: if a parentheses-enclosed group begins with a group name in angular brackets, e.g. '\(<id>[a-z][a-z0-9]*\)', 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')`.

`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 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 or 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`.

`realpat`
: 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.

3.8 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 VAX, transport the file to a Mac, and read it back there). Details of the format not explained here; read the source if you're interested.[2]

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

[2] The 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.

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 an infinite loop).

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.

3.9 Built-in module `struct`

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

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:

Format	C	Python
`'x'`	pad byte	no value
`'c'`	char	string of length 1
`'b'`	signed char	integer
`'h'`	short	integer
`'i'`	int	integer
`'l'`	long	integer
`'f'`	float	float
`'d'`	double	float

A format character may be preceded by an integral repeat count; e.g. the format string `'4h'` means exactly the same as `'hhhh'`.

C numbers are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Examples (all on a big-endian machine):

```
pack('hhl', 1, 2, 3) == '\000\001\000\002\000\000\000\003'
unpack('hhl', '\000\001\000\002\000\000\000\003') == (1, 2, 3)
calcsize('hhl') == 8
```

Hint: to align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero, e.g. the format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries.

(More format characters are planned, e.g. `'s'` for character arrays, upper case 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.)

Chapter 4

Standard Modules

The modules described in this chapter are implemented in Python, but are considered to be a part of Python's standard environment: they are always available.[1]

4.1 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. 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 a1 a2')
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
>>>
```

[1]at least in theory — it is possible to botch the library installation or to sabotage the module search path so that these modules cannot be found.

The exception `getopt.error = '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.

4.2 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 dependen 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`:

`name`
: The name of the OS dependent module imported, e.g. `'posix'` or `'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.

`execl(`*path*`,` *arg0*`,` *arg1*`,` *...*`)`

This is equivalent to a call to `os.execv` with an *argv* of `[`*arg0*`, `*arg1*`, ...]`.

`execle(`*path*`, `*arg0*`, `*arg1*`, `*...*`, `*env*`)`
This is equivalent to a call to `os.execve` with an *argv* of `[`*arg0*`, `*arg1*`, ...]`.

`execlp(`*path*`, `*arg0*`, `*arg1*`, `*...*`)`
This is like `execl` but duplicates the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from `environ['PATH']`.

`execvp(`*path*`, `*arg0*`, `*arg1*`, `*...*`)`
`execvp` is for `execv` what `execlp` is for `execl`.

4.3 Standard Module `rand`

This module implements a pseudo-random number generator with an interface similar to `rand()` in C. It defines the following functions:

`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.

4.4 Standard Module `regsub`

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

`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 '`\`*digit*' 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']`.

4.5 Standard Module `string`

This module defines some constants useful for checking character classes, some exceptions, and some useful string functions. The constants 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 exceptions are:

`atof_error`
: Exception raised by `atof` when a non-float string argument is detected. The exception argument is the offending string.

`atoi_error`
: Exception raised by `atoi` when a non-integer string argument is detected. The exception argument is the offending string.

`atol_error`
: Exception raised by `atol` when a non-integer string argument is detected. The exception argument is the offending string.

`index_error`
: Exception raised by `index` when *sub* is not found. The exception argument is undefined (it may be a tuple containing the offending arguments to `index` or it may be the constant string `'substring not found'`).

The functions are:

`atof(`*s*`)`
: 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(`*s*`)`
: Convert a string to an integer. The string must consist of one or more digits, optionally preceded by a sign (`+` or `-`).

`atol(`*s*`)`
: Convert a string to a long integer. The string must consist of one or more digits, optionally preceded by a sign (`+` or `-`).

`expandtabs(`*s*`, `*tabsize*`)`
: 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(`*s*`, `*sub*`, `*i*`)`
: Return the lowest index in *s* not smaller than *i* where the substring *sub* is found. Return `-1` when *sub* does not occur as a substring of *s* with index at least *i*. If *i* is omitted, it defaults to `0`. If *i* is negative, `len(`*s*`)` is added.

`rfind(`*s*`, `*sub*`, `*i*`)`
: Like `find` but finds the highest index.

`index(`*s*`, `*sub*`, `*i*`)`
: Like `find` but raise `index_error` when the substring is not found.

`rindex(`*s*`, `*sub*`, `*i*`)`
: Like `rfind` but raise `index_error` when the substring is not found.

`lower(`*s*`)`
: Convert letters to lower case.

`split(`*s*`)`
: Returns a list of the whitespace-delimited words of the string *s*.

`splitfields(`*s*`, `*sep*`)`
: Returns a list containing the fields of the string *s*, using the string *sep* as a separator. The list will have one more items than the number of non-overlapping occurrences of the separator in the string. Thus, `string.splitfields(`*s*`, ' ')` is not the same as `string.split(`*s*`)`, as the latter only returns non-empty words. As a special case, `splitfields(`*s*`, '')` returns `[`*s*`]`, for any string *s*. (See also `regsub.split()`.)

`join(`*words*`)`
: Concatenate a list or tuple of words with intervening spaces.

`joinfields(`*words*`, `*sep*`)`
: Concatenate a list or tuple of words with intervening separators. It is always true that `string.joinfields(string.splitfields(`*t*`, `*sep*`), `*sep*`)` equals *t*.

`strip(`*s*`)`
: Removes leading and trailing whitespace from the string *s*.

`swapcase(`*s*`)`
: Converts lower case letters to upper case and vice versa.

`upper(`*s*`)`
: Convert letters to upper case.

`ljust(`*s*`, `*width*`)`
`rjust(`*s*`, `*width*`)`
`center(`*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 *width* characters wide, created by padding the string *s* with spaces until the given width on the right, left or both sides. The string is never truncated.

`zfill(`*s*`, `*width*`)`
: Pad a numeric string on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly.

4.6 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.

Chapter 5

UNIX ONLY

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.

5.1 Built-in Module `dbm`

Dbm provides python programs with an interface to the unix `ndbm` 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. `KeyError` 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 `'rw'` as for `open`, and *filemode* is the unix mode of the file, used only when the database has to be created.

5.2 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(s) they are in.) 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 entries, in arbitrary order.

5.3 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 diguised UNIX interface). It is available in all Python versions except on the Macintosh; the MS-DOS version does not support certain functions. The descriptions below are very terse; refer to the corresponding UNIX manual entry for more information.

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. (Modifying this dictionary does not affect the string environment of the interpreter.) For example, `posix.environ['HOME']` is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

`error`
This exception is raised when an 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:

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

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

`close(`*fd*`)`
Close file descriptor *fd*.

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

`dup2(`*fd*`, `*fd2*`)`
Duplicate file descriptor *fd* 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*`)`
: 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()`.

`fdopen(`*fd*`, `*mode*`)`
: Return an open file object connected to the file descriptor *fd*, open for reading and/or writing according to the *mode* string (which has the same meaning as the *mode* argument 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.)

`getpid()`
: Return the current process id. (Not on MS-DOS.)

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

`getuid()`
: 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.

`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*`)`
: 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'` or `'w'`. (Not on MS-DOS.)

`read(`*fd*`,` *n*`)`
: Read at most *n* bytes from file descriptor *fd*. Return a string containing the bytes read.

`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.environ`, `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 `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; an 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 proces id, 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.

5.4 Standard Module `posixpath`

This module implements some useful functions on POSIX pathnames.

`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.

`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 *p* is a mount point. (This currently checks whether *p*`/..` is on a different device from *p* or whether *p*`/..` and *p* point to the same i-node on the same device — is this test correct 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.

`normcase(`*p*`)`
: Normalize the case of a pathname. This returns the path unchanged; however, a similar function in `macpath` converts upper case to lower case.

`samefile(`*p*`, `*q*`)`
: Return true if both pathname arguments refer to the same file or directory (as indicated

by device number and i-node number). Raise an exception if a stat call on either pathname fails.

`split(`*p*`)`
: Split the pathname *p* in a pair `(`*head*`, `*tail*`)`, where *tail* is the last pathname component and *head* is everything leading up to that. If *p* ends in a slash (except if it is the root), the trailing slash is removed and the operation applied to the result; otherwise, `join(`*head*`, `*tail*`)` equals *p*. The *tail* part never contains a slash. Some boundary cases: if *p* is the root, *head* equals *p* and *tail* is empty; if *p* is empty, both *head* and *tail* are empty; if *p* contains no slash, *head* is empty and *tail* equals *p*.

`splitext(`*p*`)`
: Split the pathname *p* in a pair `(`*root*`, `*ext*`)` such that *root* `+` *ext* `==` *p*, the last component of *root* contains no periods, and *ext* is empty or begins with a period.

`walk(`*p*`, `*visit*`, `*arg*`)`
: Calls the function *visit* with arguments `(`*arg*`, `*dirname*`, `*names*`)` for each directory in the directory tree rooted at *p* (including *p* itself, if it is a directory). The argument *dirname* specifies the visited directory, the argument *names* lists the files in the directory (gotten from `posix.listdir(`*dirname*`)`). The *visit* function may modify *names* to influence the set of directories visited below *dirname*, e.g., to avoid visiting certain parts of the tree. (The object referred to by *names* must be modified in place, using `del` or slice assignment.)

5.5 Built-in Module `pwd`

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

Password database entries are reported as 7-tuples containing the following items from the password database (see '`<pwd.h>`'), in order: `pw_name`, `pw_passwd`, `pw_uid`, `pw_gid`, `pw_gecos`, `pw_dir`, `pw_shell`. The uid and gid items are integers, all others are strings. An exception is raised if the entry asked for cannot be found.

It defines the following items:

`getpwuid(`*uid*`)`
: Return the password database entry for the given numeric user ID.

`getpwnam(`*name*`)`
: Return the password database entry for the given user name.

`getpwall()`
: Return a list of all available password database entries, in arbitrary order.

5.6 Built-in module `select`

This module provides access to the function `select` available in most UNIX versions. It defines the following:

`error`

The exception raised when an error occurs. 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()`.

`select(`*iwtd*`,` *owtd*`,` *ewtd*`,` *timeout*`)`

This is a straightforward interface to the UNIX `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 `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 last argument is a time-out specified as a floating point number in seconds. When the *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. `sys.stdin`, or objects returned by `open()` or `posix.popen()`), socket objects returned by `socket.socket()`, and the module `stdwin` which happens to define a function `fileno()` for just this purpose. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a UNIX file descriptor, not just a random integer).

5.7 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 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 higer-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.cwi.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 and asynchronous mode are not supported; see module `select` for a way to do non-blocking socket I/O.

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()`.

`SOCK_STREAM`
`SOCK_DGRAM`
: These constants represent the socket types, used for the second argument to `socket()`. (There are other types, but only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

`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.

`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.

`fromfd(`*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).

5.7.1 Socket Object Methods

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 an address. The socket must not already be bound.

`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.

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

`getpeername()`
: Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IP socket, for instance.

`getsockname()`
: Return the socket's own address. This is useful to find out the port number of an IP socket, for instance.

`getsockopt(`*level*`, `*optname*`, `*buflen*`)`
: Return the value of the given socket option (see the UNIX man page *getsockopt*(2)). The needed symbolic constants are defined in module SOCKET. If the optional third argument 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's 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 argument (in the range 0-5) specifies the maximum number of queued connections.

`makefile(`*mode*`)`
: Return a *file object* associated with the socket. (File objects were described earlier under Built-in Types.) The file object references a `dup`ped version of the socket file descriptor, so the file object and socket object may be closed or garbage-collected independently.

`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*`)`
: 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.

`send(`*string*`)`
: Send data to the socket. The socket must be connected to a remote socket.

`sendto(`*string*`, `*address*`)`
: Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by `address`.

`setsockopt(`*level*`, `*optname*`, `*value*`)`
: Set the value of the given socket option (see the UNIX man page *setsockopt*(2)). The needed symbolic constants are defined in module `SOCKET`. 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.

5.7.2 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(0)
conn, addr = s.accept()
print 'Connected by', addr
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)
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', `data`
```

5.8 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 and Sun Sparc systems only.

It defines the following constant and functions:

`error`
: Raised on thread-specific errors.

`start_new_thread(`*func*`, `*arg*`)`
: Start a new thread. The thread executes the function *func* with the argument list *arg* (which must be a tuple). When the function returns, the thread silently exits. When the function raises terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

`exit_thread()`
: Exit the current thread silently. Other threads continue to run. **Caveat:** code in pending `finally` clauses is not executed.

`exit_prog(`*status*`)`
: Exit all threads and report the value of the integer argument *status* as the exit status of the entire program. **Caveat:** code in pending `finally` clauses, in this thread or in other threads, is not executed.

`allocate_lock()`
: Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

Lock objects have the following methods:

`acquire(`*waitflag*`)`
: 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 `None`. If the integer *waitflag* 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 1 if the lock is acquired successfully, 0 if not.

`release()`
: Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

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

Caveats:

- Threads interact strangely with interrupts: the `KeyboardInterrupt` exception will be received by an arbitrary thread.

- Calling `sys.exit(`*status*`)` or executing `raise SystemExit,` *status* is almost equivalent to calling `thread.exit_prog(`*status*`)`, except that the former ways of exiting the entire program do honor `finally` clauses in the current thread (but not in other threads).

- Not all built-in functions that may block waiting for I/O allow other threads to run, although the most popular ones (`sleep`, `read`, `select`) work as expected.

Chapter 6

MULTIMEDIA EXTENSIONS

The modules described in this chapter implement various algorithms that are mainly useful for multimedia applications. They are available at the discretion of the installation.

6.1 Built-in module `audioop`

The audioop module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples of 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*`)`
: This function returns a fragment that 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*`)`
: This routine decodes an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm` for details on ADPCM coding. The routine returns a tuple `(`*sample* `,` *newstate*`)` where the sample has the width specified in *width*.

`adpcm32lin(`*adpcmfragment* `,` *width* `,` *state*`)`
: This routine decodes an alternative 3-bit ADPCM code. See `lin2adpcm3` for details.

`avg(`*fragment* `,` *width*`)`
: This function returns the average over all samples in the fragment.

`avgpp(`*fragment* `,` *width*`)`
: This function returns the average peak-peak value over all samples in the fragment. No

filtering is done, so the useability of this routine is questionable.

`bias(`*fragment*`,` *width*`,` *bias*`)`
: This function returns a fragment that is the original fragment with a bias added to each sample.

`cross(`*fragment*`,` *width*`)`
: This function returns the number of zero crossings in the fragment passed as an argument.

`findfactor(`*fragment*`,` *reference*`)`
: This routine (which only accepts 2-byte sample fragments) calculates a factor F such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e. it calculates the factor with which you should multiply *reference* to make it match as good as possible to *fragment*. The fragments should be the same size.

  The time taken by this routine is proportional to `len(fragment)`.

`findfit(`*fragment*`,` *reference*`)`
: This routine (which only accepts 2-byte sample fragments) tries to match *reference* as good as possible to a portion of *fragment* (which should be the longer fragment). It (conceptually) does this by taking slices out of *fragment*, using `findfactor` to compute the best match, and minimizing the result. It returns a tuple `(`*offset*`,` *factor*`)` with offset the (integer) offset into *fragment* where the optimal match started and *factor* the floating-point factor as per findfactor.

`findmax(`*fragment*`,` *length*`)`
: This routine (which only accepts 2-byte sample fragments) searches *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e. it returns i for which `rms(fragment[i*2:(i+length)*2])` is maximal.

  The routine takes time proportional to `len(fragment)`.

`getsample(`*fragment*`,` *width*`,` *index*`)`
: This function returns the value of sample *index* from the fragment.

`lin2lin(`*fragment*`,` *width*`,` *newwidth*`)`
: This function converts samples between 1-, 2- and 4-byte formats.

`lin2adpcm(`*fragment*`,` *width*`,` *state*`)`
: This function converts samples to 4 bit Intel/DVI 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/DVI ADPCM algorythm has been selected for use by the IMA, so may well become a standard.

  `State` is a tuple containing the state of the coder. The coder returns a tuple `(`*adpcmfrag*`,` *newstate*`)`, and the *newstate* should be passed to the next call of lin2adpcm. In the initial call `None` can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

`lin2adpcm3(`*fragment*`,` *width*`,` *state*`)`
: This is an alternative ADPCM coder that uses only 3 bits per sample. It is not compatible with the Intel/DVI ADPCM coder and its output is not packed (due to laziness on the side of the author). Its use is discouraged.

`lin2ulaw(`*fragment*`, `*width*`)`
: This function converts samples in the audio fragment to U-LAW encoding and returns 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(`*fragment*`, `*width*`)`
: This function returns a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

`max(`*fragment*`, `*width*`)`
: This function returns the maximum of the *absolute value* of all samples in a fragment.

`maxpp(`*fragment*`, `*width*`)`
: This function returns the maximum peak-peak value in the sound fragment.

`mul(`*fragment*`, `*width*`, `*factor*`)`
: Mul returns a fragment that has all samples in the original framgent multiplied by the floating-point value *factor*. Overflow is silently ignored.

`reverse(`*fragment*`, `*width*`)`
: This function reverses the samples in a fragment and returns the modified fragment.

`tomono(`*fragment*`, `*width*`, `*lfactor*`, `*rfactor*`)`
: This function converts a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

`tostereo(`*fragment*`, `*width*`, `*lfactor*`, `*rfactor*`)`
: This function generates a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

`mul(`*fragment*`, `*width*`, `*factor*`)`
: Mul returns a fragment that has all samples in the original framgent multiplied by the floating-point value *factor*. Overflow is silently ignored.

`rms(`*fragment*`, `*width*`, `*factor*`)`
: Returns the root-mean-square of the fragment, i.e.

$$\sqrt{\frac{\sum {S_i}^2}{n}}$$

This is a measure of the power in an audio signal.

`ulaw2lin(`*fragment*`, `*width*`)`
: This function converts sound fragments in ULAW encoding to linearly encoded sound fragments. ULAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as `mul` or `max` make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(sample, width, lfactor)
    rsample = audioop.mul(sample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

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 lin2adpcm) along to the decoder, not the final state (as returned by the coder). If you want to use `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 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 `find...` routines might look a bit funny at first sight. They are primarily meant for doing 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, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[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 = prefill + audioop.mul(outputdata,2,-factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

6.2 Built-in module `imageop`

The imageop module contains some useful operations on images. It operates on images consisting of 8 or 32 bit pixels stored in python strings. This is the same format as used by `gl.lrectwrite` 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*`)`
: This function takes the image in `image`, which should by `width` by `height` in size and consist of pixels of `psize` bytes, and returns the selected part of that image. `X0`, `y0`, `x1` and `y1` are like the `lrectread` parameters, i.e. 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*`)`
: This function returns a `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*`)`
: This function runs 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.

`grey2mono(`*image*`,` *width*`,` *height*`,` *threshold*`)`
: This function converts a 8-bit deep greyscale image to a 1-bit deep image by tresholding all the pixels. The resulting image is tightly packed and is probably only useful as an argument to `mono2grey`.

`dither2mono(`*image*`,` *width*`,` *height*`)`
: This function also converts an 8-bit greyscale image to a 1-bit monochrome image but it uses a (simple-minded) dithering algorithm.

`mono2grey(`*image*`,` *width*`,` *height*`,` *p0*`,` *p1*`)`
: This function converts 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.

`grey2grey4(`*image*`,` *width*`,` *height*`)`
: Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

`grey2grey2(`*image*`,` *width*`,` *height*`)`
: Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

`dither2grey2(`*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.

`grey42grey(`*image*`,` *width*`,` *height*`)`
: Convert a 4-bit greyscale image to an 8-bit greyscale image.

`grey22grey(`*image*`,` *width*`,` *height*`)`

Convert a 2-bit greyscale image to an 8-bit greyscale image.

6.3 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 Indepent 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 immedeately be passed to compress. Currently only 1 byte and 4 byte pixels are allowed, the former being treaded 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.

  `'smooth'`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.

6.4 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.

`longimagedata(`*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.

`longstoimage(`*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 images 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 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.

Chapter 7

CRYPTOGRAPHIC EXTENSIONS

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation.

7.1 Built-in module `md5`

This module implements the interface to RSA's MD5 message digest algorithm (see also the file '`md5.doc`'). It's use is very straightforward: use the function `md5` to create an *md5*-object. You can now "feed" this object with arbitrary strings.

At any time you can ask the "final" digest of the object. Internally, a temorary copy of the object is made and the digest is computed and returned. Because of the copy, the digest operation is not desctructive for the object. Before a more exact description of the use, a small example: to obtain the digest of the string `'abc'`, use ...

```
>>> from md5 import md5
>>> m = md5()
>>> m.update('abc')
>>> m.digest()
'\220\001P\230<\322O\260\326\226?}(\341\177r'
```

More condensed:

```
>>> md5('abc').digest()
'\220\001P\230<\322O\260\326\226?}(\341\177r'
```

`md5(`*arg*`)`

Create a new md5-object. *arg* is optional: if present, an initial `update` method is called with *arg* as argument.

An md5-object has the following methods:

`update(`*arg*`)`
  Update this md5-object with the string *arg*.

`digest()`
  Return the *digest* of this md5-object. Internally, a copy is made and the C-function `MD5Final` is called. Finally the digest is returned.

`copy()`
  Return a separate copy of this md5-object. An `update` to this copy won't affect the original object.

7.2 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 *and*s, *invert*s and *or*s, 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 excact description). An mpz-number is printed like this: `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 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.

`powm(`*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*.

`gcdext(`*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.

7.3 Built-in module `rotor`

This module implements a rotor-based encryption algorithm, contributed by Lance Ellinghouse. Currently no further documentation is available — you are kindly advised to read the source...

Chapter 8

STDWIN ONLY

8.1 Built-in Module `stdwin`

This module defines several new object types and functions that provide access to the functionality of the Standard Window System Interface, STDWIN [CWI report CR-R8817]. It is available on systems to which STDWIN has been ported (which is most systems). It is only available if the `DISPLAY` environment variable is set or an explicit '`-display` *displayname*' argument is passed to the 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).

8.1.1 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.[1]

`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 `()`.

[1]The Python version of STDWIN does not support draw procedures; all drawing requests are reported as draw events.

`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.

`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.

`getdefscrollbars()`
: Return the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll bars.

`getdefwinpos()`
: Return the default window position for windows opened subsequently.

`getdefwinsize()`
: 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 wheter you are on a machine that supports more than two colors:

```
if stdwin.fetchcolor('black') <> \
          stdwin.fetchcolor('red') <> \
          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`, `textbreak`, `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. Note: normally, menus are created locally; see the window method `menucreate` 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.

`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.

`askync(`*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 `stdwinevents`. 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 STDWIN 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 curent 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 implementation. (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 STDWIN, 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.

8.1.2 Window Object Methods

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*`)`
: Invalidate 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()`.

`getorigin()`
: 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.

`setorigin(`*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 applications 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.

`setwincursor(`*name*`)`
: Set the window cursor to a cursor of the given name. It raises the `RuntimeError` exception if no cursor of the given name exists. Suitable names include `'ibeam'`, `'arrow'`, `'cross'`, `'watch'` and `'plus'`. On X11, there are many more (see '`<X11/cursorfont.h>`').

`setwinpos(`*h*`, `*v*`)`
: Set the the 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_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.

8.1.3 Drawing Object Methods

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*`), (`*a1*`, `*a2*`))`
: Draw an elliptical arc with given center point. `(`*rh*`, `*rv*`)` gives the half sizes of the horizontal and vertical radii. `(`*a1*`, `*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*`), (`*a1*`, `*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*`)`
: Fill 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*`), (`*a1*`, `*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()`
`textbreak()`
`textwidth()`
: These functions are similar to the corresponding functions described above for the

`stdwin` module, but affect or use the current drawing font instead of the global default font. When a drawing object is created, its font is set to the window's default font, which is in turn initialized from the global default font when the window is created.

`bitmap(`*point*`,` *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).

`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. clipping region

`noclip()`
: Reset the clipping region to the entire window.

`close()`
`enddrawing()`
: Discard the drawing object. It should not be used again.

8.1.4 Menu Object Methods

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 disables 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.

8.1.5 Bitmap Object Methods

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. The following methods are defined:

`getsize()`
: Return a tuple representing the width and height of the bitmap. (This returns the values that have been passed to the `newbitmap` function.)

`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.

8.1.6 Text-edit Object Methods

A text-edit object represents a text-edit block. For semantics, see the STDWIN documentation for C programmers. The following methods exist:

`arrow(`*code*`)`
: Pass an arrow event to the text-edit block. The *code* must be one of `WC_LEFT`, `WC_RIGHT`, `WC_UP` or `WC_DOWN` (see module `stdwinevents`).

`draw(`*rect*`)`
: Pass a draw event to the text-edit block. The rectangle specifies the redraw area.

`event(`*type*`, `*window*`, `*detail*`)`
: Pass an event gotten from `stdwin.getevent()` to the text-edit block. Return true if the event was handled.

`getfocus()`
: Return 2 integers representing the start and end positions of the focus, usable as slice indices on the string returned by `gettext()`.

`getfocustext()`
: Return the text in the focus.

`getrect()`
: Return a rectangle giving the actual position of the text-edit block. (The bottom coordinate may differ from the initial position because the block automatically shrinks or grows to fit.)

`gettext()`
: Return the entire text buffer.

`move(`*rect*`)`
: Specify a new position for the text-edit block in the document.

`replace(`*str*`)`
: Replace the text in the focus by the given string. The new focus is an insert point at the end of the string.

`setfocus(`*i*`, `*j*`)`
: Specify the new focus. Out-of-bounds values are silently clipped.

`settext(`*str*`)`
: Replace the entire text buffer by the given string and set the focus to `(0, 0)`.

`setview(`*rect*`)`
: Set the view rectangle to *rect*. If *rect* is `None`, viewing mode is reset. In viewing mode, all output from the text-edit object is clipped to the viewing rectangle. This may be useful to implement your own scrolling text subwindow.

`close()`
: Discard the text-edit object. It should not be used again.

8.1.7 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.

```
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()
```

8.2 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

```
>>> from stdwinevents import *
>>>
```

8.3 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:

```
>>> 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 `(`*left*`, `*top*`), (`*right*`, `*bottom*`)` is empty if $left \geq right$ or $top \geq bottom$.

`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 `(`*left*`, `*top*`), (`*right*`, `*bottom*`)` if $left \leq h < right$ and $top \leq v < bottom$.

`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: `(`*left*`, `*top*`), (`*width*`, `*height*`).`

`geom2rect(`*geom*`)`

Converts a rectangle given in geometry representation back to the standard rectangle representation `(`*left*`, `*top*`), (`*right*`, `*bottom*`).`

Chapter 9

SGI IRIX ONLY

The modules described in this chapter provide interfaces to features that are unique to SGI's IRIX operating system (versions 4 and 5).

9.1 Built-in Module `al`

This module provides access to the audio facilities of the Indigo and 4D/35 workstations, described in section 3A of the IRIX 4.0 man pages (and also available as an option in IRIX 3.3). You'll need to read those man pages to understand what these functions do! Some of the functions are not available in releases below 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

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.)

Module `al` defines the following functions:

`openport(`*name*`, `*direction*`, `*config*`)`
: Equivalent to the C function ALopenport(). The name and direction arguments are strings. The optional config argument is an opaque configuration object as returned by `al.newconfig()`. The return value is an opaque port object; methods of port objects are described below.

`newconfig()`
: Equivalent to the C function ALnewconfig(). The return value is a new opaque configuration object; methods of configuration objects are described below.

`queryparams(`*device*`)`
: Equivalent to the C function ALqueryparams(). The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

`getparams(`*device*`,` *list*`)`
: Equivalent to the C function ALgetparams(). 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*`)`
: Equivalent to the C function ALsetparams(). The device argument is an integer.The list argument is a list such as returned by `al.queryparams`.

Configuration objects (returned by `al.newconfig()` have the following methods:

`getqueuesize()`
: Return the queue size; equivalent to the C function ALgetqueuesize().

`setqueuesize(`*size*`)`
: Set the queue size; equivalent to the C function ALsetqueuesize().

`getwidth()`
: Get the sample width; equivalent to the C function ALgetwidth().

`getwidth(`*width*`)`
: Set the sample width; equivalent to the C function ALsetwidth().

`getchannels()`
: Get the channel count; equivalent to the C function ALgetchannels().

`setchannels(`*nchannels*`)`
: Set the channel count; equivalent to the C function ALsetchannels().

`getsampfmt()`
: Get the sample format; equivalent to the C function ALgetsampfmt().

`setsampfmt(`*sampfmt*`)`
: Set the sample format; equivalent to the C function ALsetsampfmt().

`getfloatmax()`
: Get the maximum value for floating sample formats; equivalent to the C function ALgetfloatmax().

`setfloatmax(`*floatmax*`)`
: Set the maximum value for floating sample formats; equivalent to the C function ALsetfloatmax().

Port objects (returned by `al.openport()` have the following methods:

`closeport()`
: Close the port; equivalent to the C function ALcloseport().

`getfd()`
: Return the file descriptor as an int; equivalent to the C function ALgetfd().

`getfilled()`
: Return the number of filled samples; equivalent to the C function ALgetfilled().

`getfillable()`
: Return the number of fillable samples; equivalent to the C function ALgetfillable().

`readsamps(`*nsamples*`)`

Read a number of samples from the queue, blocking if necessary; equivalent to the C function ALreadsamples. The data is returned 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.

`writesamps(`*samples*`)`
: Write samples into the queue, blocking if necessary; equivalent to the C function ALwritesamples. The samples are encoded as described for the `readsamps` return value.

`getfillpoint()`
: Return the 'fill point'; equivalent to the C function ALgetfillpoint().

`setfillpoint(`*fillpoint*`)`
: Set the 'fill point'; equivalent to the C function ALsetfillpoint().

`getconfig()`
: Return a configuration object containing the current configuration of the port; equivalent to the C function ALgetconfig().

`setconfig(`*config*`)`
: Set the configuration from the argument, a configuration object; equivalent to the C function ALsetconfig().

`getstatus(`*list*`)`
: Get status information on last error equivalent to C function ALgetstatus().

9.2 Standard Module AL

This module defines symbolic constants needed to use the built-in module `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 *
```

9.3 Built-in Module fl

This module provides an interface to the FORMS Library by Mark Overmars, version 2.0b. For more info about FORMS, write to `markov@cs.ruu.nl`.

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 button are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions `fl_addto_form` and `fl_end_form`, and the equivalent of `fl_bgn_form` is called `fl.make_form`.

Watch out for the somewhat confusing terminology: FORMS uses the word *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 avaiable, though, so you can mix FORMS with pure GL windows.

Please note: importing `fl` implies a call to the GL function `foreground()` and to the FORMS routine `fl_init()`.

9.3.1 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.EVENT`.

`check_forms()`
: Check for FORMS events. Returns what `do_forms` above returns, or `None` if there is no event that immediately needs interaction.

`set_event_call_back(`*function*`)`
: Set the event callback function.

`set_graphics_mode(`*rgbmode*`,` *doublebuffering*`)`
: Set the graphics modes.

`get_rgbmode()`
: Return the current rgb mode. This is the value of the C global variable `fl_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`). The *but2* and *but3* arguments are optional.

`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 inm 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.

`qdevice(`*dev*`)`
`unqdevice(`*dev*`)`
`isqueued(`*dev*`)`
`qtest()`
`qread()`
`qreset()`
`qenter(`*dev*`,` *val*`)`
`get_mouse()`
`tie(`*button*`,` *valuator1*`,` *valuator2*`)`
: 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_forms()` 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`.

9.3.2 Form object methods and data attributes

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()`
Hide the form.

`redraw_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.

`end_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_clock(`*type*`, `*x*`, `*y*`, `*w*`, `*h*`, `*name*`)`
Add a clock object to the form.
Method: `get_clock`.

`add_button(`*type*`, `*x*`, `*y*`, `*w*`, `*h*`, `*name*`)`
Add a button object to the form.
Methods: `get_button`, `set_button`.

`add_lightbutton(`*type*`, `*x*`, `*y*`, `*w*`, `*h*`, `*name*`)`
Add a lightbutton object to the form.
Methods: `get_button`, `set_button`.

`add_roundbutton(`*type*`, `*x*`, `*y*`, `*w*`, `*h*`, `*name*`)`
Add a roundbutton object to the form.
Methods: `get_button`, `set_button`.

`add_slider(`*type*`, `*x*`, `*y*`, `*w*`, `*h*`, `*name*`)`

Add a slider object to the form.
Methods: `set_slider_value`, `get_slider_value`, `set_slider_bounds`, `get_slider_bounds`, `set_slider_return`, `set_slider_size`, `set_slider_precision`, `set_slider_step`.

`add_valslider(`*type*, *x*, *y*, *w*, *h*, *name*`)`
Add a valslider object to the form.
Methods: `set_slider_value`, `get_slider_value`, `set_slider_bounds`, `get_slider_bounds`, `set_slider_return`, `set_slider_size`, `set_slider_precision`, `set_slider_step`.

`add_dial(`*type*, *x*, *y*, *w*, *h*, *name*`)`
Add a dial object to the form.
Methods: `set_dial_value`, `get_dial_value`, `set_dial_bounds`, `get_dial_bounds`.

`add_positioner(`*type*, *x*, *y*, *w*, *h*, *name*`)`
Add a positioner object to the form.
Methods: `set_positioner_xvalue`, `set_positioner_yvalue`, `set_positioner_xbounds`, `set_positioner_ybounds`, `get_positioner_xvalue`, `get_positioner_yvalue`, `get_positioner_xbounds`, `get_positioner_ybounds`.

`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 a 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`, `replace_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`, `deselect_browser`, `isselected_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:

Name	Type	Meaning
`window`	int (read-only)	GL window id
`w`	float	form width
`h`	float	form height
`x`	float	form x origin
`y`	float	form y origin
`deactivated`	int	nonzero if form is deactivated
`visible`	int	nonzero if form is visible
`frozen`	int	nonzero if form is frozen
`doublebuf`	int	nonzero if double buffering on

9.3.3 FORMS object methods and data attributes

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.

`delete_object()`
: Delete the object.

`show_object()`
: Show the object.

`hide_object()`
: Hide the object.

`redraw_object()`
: Redraw the object.

`freeze_object()`
: Freeze the object.

`unfreeze_object()`
: Unfreeze the object.

FORMS objects have these data attributes; see the FORMS documentation:

Name	Type	Meaning
`objclass`	int (read-only)	object class
`type`	int (read-only)	object type
`boxtype`	int	box type
`x`	float	x origin
`y`	float	y origin
`w`	float	width
`h`	float	height
`col1`	int	primary color
`col2`	int	secondary color
`align`	int	alignment
`lcol`	int	label color
`lsize`	float	label font size
`label`	string	label string
`lstyle`	int	label style
`pushed`	int (read-only)	(see FORMS docs)
`focus`	int (read-only)	(see FORMS docs)
`belowmouse`	int (read-only)	(see FORMS docs)
`frozen`	int (read-only)	(see FORMS docs)
`active`	int (read-only)	(see FORMS docs)
`input`	int (read-only)	(see FORMS docs)
`visible`	int (read-only)	(see FORMS docs)
`radio`	int (read-only)	(see FORMS docs)
`automatic`	int (read-only)	(see FORMS docs)

9.4 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:

```
import fl
from FL import *
```

9.5 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 `fl` 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!

9.6 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*`)`.

`getstrwidth(`*string*`)`
: Returns the width, in pixels, of the string when drawn in this font. Calls

`fmgetstrwidth(`*fh* `,` *string*`)`.

9.7 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

  ```
  lmdef(deftype, index, np, props)
  ```

  is translated to Python as

  ```
  lmdef(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

  ```
  getmcolor(i, &red, &green, &blue)
  ```

  is translated to Python as

```
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.

`nurbssurface(`*s_k*`, `*t_k*`, `*ctl*`, `*s_ord*`, `*t_ord*`, `*type*`)`
: Defines a nurbs surface. The dimensions of *ctl*`[][]` 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*`)`
`select(`*n*`)`
: The only argument to these functions specifies the desired size of the pick or select buffer.

`endpick()`
`endselect()`
: 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:

```
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()
```

9.8 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.

9.9 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 image 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 10

SUNOS ONLY

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).

10.1 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.

10.1.1 Audio device object methods

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 (due to buffering of up to one second of sound).

`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 number 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 audio man page. Member names are slightly different from their C counterparts: a status object is only a single structure. Members of the `play` substructure have '`o_`' prepended to their name and members of the `record` structure have '`i_`'. 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`.

`ibufcount()`
: 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 immedeately return, otherwise it will block.

There is a companion module, `SUNAUDIODEV`, 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. This is because that feature makes heavy use of signals, and these do not map too well onto Python.

Index