Python Library Reference (DRAFT)

Guido van Rossum
Dept. CST, CWI, Kruislaan 413
1098 SJ Amsterdam, The Netherlands
E-mail: gu...@cwi.nl

August 31, 2023

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 Tutorial document. The Language Reference document (XXX not yet existing) gives a more formal reference to the language.

Contents

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.

2 Built-in Types, Exceptions and Functions

Names for built-in exceptions and functions are found in a separate read-only symbol table which cannot be modified. 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 by syntactic constructs (such as constants) or built-in functions. 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.

2.1.1 Numeric Types

There are two numeric types: integers and floating point numbers. Integers are implemented using `long` in C, so they have at least 32 bits of precision. Floating point numbers are implemented using `double` in C. All bets on precision are off. Numbers are created by numeric constants or as the result of built-in functions and operators.

Numeric types support the following operations:

Operation	Result	Notes
`abs`(x)	absolute value of x	
`int`(x)	x converted to 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	(3)

Notes:

(1) This may round or truncate as in C; see functions `floor` and `ceil` in module `math`.

(2) Integer division is defined as in C: the result is an integer; with positive operands, it truncates towards zero; with a negative operand, the result is unspecified.

(3) Only defined for integers.

Mixed arithmetic is not supported; both operands must have the same type. Mixed comparisons return the wrong result (floats always compare smaller than integers).[2]

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

[2] These restrictions are bugs in the language definitions and will be fixed in the future.

2.1.2 Sequence Types

There are three sequence types: strings, lists and tuples. Strings constants are written in single quotes: `'xyzzy'`. Lists are constructed with square brackets: `[a, b, c]`. Tuples are constructed by the comma operator or with an empty set of parentheses: `a, b, c` or `()`.

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	true if an item of s is equal to x	
x `not in` s	false if an item of s is equal to x	
s`+`t	the concatenation of s and t	
s`*`n, n`*`s	n copies of s concatenated	(1)
$s[i]$	i'th item of s	
$s[i{:}j]$	slice of s from i to j	(2)

Notes:

(1) Sequence repetition is only supported for strings.

(2) The slice of s from i to j is defined as the sequence of items with index k such that $i \leq k < j$. Special rules apply for negative and omitted indices; see the Tutorial or the Reference Manual.

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
$s[i] = x$	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.`insert`(i, x)	same as $s[i{:}i] = [x]$
s.`sort`()	the items of s are permuted to satisfy $s[i] \leq s[j]$ for $i < j$

2.1.3 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 strings. An empty dictionary is created by the expression `{}`. An extension of this notation is used to display dictionaries when written (see the example below).

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 elements in a	
$a[k]$	the item of a with key k	
$a[k] = x$	set $a[k]$ to x	
`del` $a[k]$	remove $a[k]$ from a	
a.`keys`()	a copy of a's list of keys	(1)
a.`has_key`(k)	true if a has a key k	

Notes:

(1) Keys are listed in random order.

A small example using a dictionary:

```
>>> tel = {}
>>> tel['jack'] = 4098
>>> tel['sape'] = 4139
>>> tel['guido'] = 4127
>>> tel['jack']
4098
>>> tel
{'sape': 4139; 'guido': 4127; 'jack': 4098}
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127; 'irv': 4127; 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
>>>
```

2.1.4 Other Built-in Types

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

Modules. The only operation on a module is member acces: $m.name$, where m is a module and *name* accesses a name defined in m's symbol table. Module members can be assigned to.

Classes and Class Objects. XXX Classes will be explained at length in a later version of this document.

Functions. Function objects are created by function definitions. The only operation on a function object is to call it: *func*(*optional-arguments*).

Built-in functions have a different type than user-defined functions, but they support the same operation.

Methods. Methods are functions that are called using the member acces notation. There are two flavors: built-in methods (such as `append()` on lists) and class member methods. Built-in methods are described with the types that support them. XXX Class member methods will be described in a later version of this document.

Type Objects. Type objects represent the various object types. An object's type is accessed by the built-in function `type()`. There are no operations on type objects.

The Null Object. This object is returned by functions that don't explicitly return a value. It supports no operations. There is exactly one null object, named `None` (a built-in name).

File Objects. File objects are implemented using C's *stdio* package and can be created with the built-in function `open()`. They have the following methods:

close() Closes the file. A closed file cannot be read or written anymore.

read(size) Reads at most `size` bytes from the file (less if the read hits EOF). The bytes are returned as a string object. An empty string is returned when EOF is hit immediately. (For certain files, like ttys, it makes sense to continue reading after an EOF is hit.)

readline(size) Reads a line of at most `size` bytes from the file. A trailing newline character, if present, is kept in the string. The size is optional and defaults to a large number (but not infinity). EOF is reported as by `read()`.

write(str) Writes a string to the file. Returns no value.

2.2 Built-in Exceptions

The following exceptions can be generated by the interpreter or built-in functions. Except where mentioned, they have a string argument (also known as the 'associated value' of an exception) indicating the detailed cause of the error. The strings listed with the exception names are their values when used in an expression or printed.

EOFError = `'end-of-file read'`
(No argument.) Raised when a built-in function (`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.)

KeyboardInterrupt = `'end-of-file read'`
(No argument.) 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.

MemoryError = `'out of memory'`
Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects).

NameError = `'undefined name'`
Raised when a name is not found. This applies to unqualified names, module names (on `import`), module members and object methods. The string argument is the name that could not be found.

RuntimeError = `'run-time error'`
Raised for a variety of reasons, e.g., division by zero or index out of range.

SystemError = `'system error'`
Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope.

TypeError = `'type error'`
Raised when an operation or built-in function is applied to an object of inappropriate type.

2.3 Built-in Functions

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

abs(x) Returns the absolute value of a number. The argument may be an integer or floating point number.

chr(i) Returns 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()`.

dir() Without arguments, this function returns the list of names in the current local symbol table, sorted alphabetically. With a module object as argument, it returns the sorted list of names in that module's global symbol table. For example:

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

divmod(a, b) Takes two integers as arguments and returns a pair of integers consisting of their quotient and remainder. For

```
q, r = divmod(a, b)
```

the invariants are:

```
a = q*b + r
abs(r) < abs(b)
r has the same sign as b
```

For example:

```
>>> divmod(100, 7)
(14, 2)
>>> divmod(-100, 7)
(-15, 5)
>>> divmod(100, -7)
(-15, -5)
>>> divmod(-100, -7)
(14, -2)
>>>
```

eval(s) Takes a string as argument and parses and evaluates it as a Python expression. The expression is executed using the current local and global symbol tables. Syntax errors are reported as exceptions. For example:

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

exec(s) Takes a string as argument and parses and evaluates it as a sequence of Python statements. The string should end with a newline (`'\n'`). The statement is executed using the current local and global symbol tables. Syntax errors are reported as exceptions. For example:

```
>>> x = 1
>>> exec('x = x+1\n')
>>> x
2
>>>
```

float(x) Converts a number to floating point. The argument may be an integer or floating point number.

input(s) Equivalent to `eval(raw_input(s))`. As for `raw_input()`, the argument is optional.

int(x) Converts a number to integer. The argument may be an integer or floating point number.

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

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

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

open(name, mode) Returns a file object (described earlier under Built-in Types). The string arguments are the same as for stdio's `fopen()`: `'r'` opens the file for reading, `'w'` opens it for writing (truncating an existing file), `'a'` opens it for appending.[3]

ord(c) Takes a string of one character and returns its ASCII value, e.g., `ord('a')` returns the integer `97`. This is the inverse of `chr()`.

range() This is a versatile function to create lists containing arithmetic progressions of integers. With two integer arguments, it returns the ascending sequence of integers starting at the first and ending one before the second argument. A single argument is used as the end point of the sequence, with 0 used as the starting point. A third argument specifies the step size; negative steps are allowed and work as expected, but don't specify a zero step. The resulting list may be empty. For example:

[3]This function should go into a built-in module `io`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 1+10)
[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(s) The 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. EOF is reported as an exception. For example:

```
>>> raw_input('Type anything: ')
Type anything: Mutant Teenage Ninja Turtles
'Mutant Teenage Ninja Turtles'
>>>
```

reload(module) Causes an already imported module to be re-parsed and re-initialized. This is useful if you have edited the module source file and want to try out the new version without leaving Python.

type(x) Returns the type of an object. Types are objects themselves: the type of a type object is its own type.

3 Built-in Modules

The modules described in this section are built into the interpreter. They must be imported using `import`. Some modules are not always available; it is a configuration option to provide them. Details are listed with the descriptions, but the best way to see if a module exists in a particular implementation is to attempt to import it.

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(T) he 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.

exit(n) Exits from Python with numeric exit status `n`. This closes all open files and performs other cleanup-actions required by the interpreter (but *finally clauses* of `try` statements are not executed!).

modules(G) ives 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(S) trings 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 `'... '`.

stdin, stdout, stderr(F) ile 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 its error messages are written to stderr. Assigning to `sys.stderr` has no effect on the interpreter; it can be used to write error messages to stderr using `print`.

3.2 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)`, `log(x)`, `log10(x)`, `pow(x,y)`, `sin(x)`, `sinh(x)`, `sqrt(x)`, `tan(x)`, `tanh(x)`.

It also defines two mathematical constants: `pi` and `e`.

3.3 Built-in Module `time`

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

sleep(secs) Suspends execution for the given number of seconds.

time() Returns the time in seconds since the Epoch (Thursday January 1, 00:00:00, 1970 UCT on UNIX machines).

In some versions (Amoeba, Mac) the following functions also exist:

millisleep(msecs) Suspends execution for the given number of milliseconds.

millitimer() Returns 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 granularity of the milliseconds functions may be more than a millisecond (100 msecs on Amoeba, 1/60 sec on the Mac).

3.4 Built-in Module `regexp`

This module provides a regular expression matching operation. It is always available.

The module defines a function and an exception:

compile(pattern) Compile a regular expression given as a string into a regular expression object. The string must be an egrep-style regular expression; this means that the characters `'('` `')'` `'*'` `'+'` `'?'` `'|'` `'^'` `'$'` are special. (It is implemented using Henry Spencer's regular expression matching functions.)

excitemerrorregexp.error

Exception raised when a string passed to `compile()` is not a valid regular expression (e.g., unmatched parentheses) or when some other error occurs during compilation or matching ("no match found" is not an error).

Compiled regular expression objects support a single method:

exec(str) Find the first occurrence of the compiled regular expression in the string `str`. The return value is a tuple of pairs specifying where a match was found and where matches were found for subpatterns specified with `'('` and `')'` in the pattern. If no match is found, an empty tuple is returned; otherwise the first item of the tuple is a pair of slice indices into the search string giving the match found. If there were any subpatterns in the pattern, the returned tuple has an additional item for each subpattern, giving the slice indices into the search string where that subpattern was found.

For example:

```
>>> import regexp
>>> r = regexp.compile('--(.*)--')
>>> s = 'a--b--c'
>>> r.exec(s)
((1, 6), (3, 4))
>>> s[1:6] # The entire match
'--b--'
>>> s[3:4] # The subpattern
'b'
>>>
```

3.5 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. Errors are reported exceptions. It defines the following items:

chdir(path) Changes the current directory to `path`.

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

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 = `'posix.error'`
The exception raised when an POSIX function returns an error. The value accompanying this exception is a pair containing the numeric error code from `errno` and the corresponding string, as would be printed by the C function `perror()`.

getcwd() Returns a string representing the current working directory.

link(src, dst) Creates a hard link pointing to `src` named `dst`.

listdir(path) Returns 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.

mkdir(path, mode) Creates a directory named `path` with numeric mode `mode`.

rename(src, dst) Renames the file or directory `src` to `dst`.

rmdir(path) Removes the directory `path`.

stat(path) Performs 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.

system(command) Executes 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()`.

umask(mask) Sets the current numeric umask and returns the previous umask.

unlink(path) Unlinks the file `path`.

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

The following functions are only available on systems that support symbolic links:

lstat(path) Like `stat()`, but does not follow symbolic links.

readlink(path) Returns a string representing the path to which the symbolic link points.

symlink(src, dst) Creates a symbolic link pointing to `src` named `dst`.

3.6 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).

3.6.1 Functions Defined in Module `stdwin`

The following functions are defined in the `stdwin` module:

open(title) Opens a new window whose initial title is given by the string argument. Returns a window object; window object methods are described below.[4]

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

getevent() Waits for and returns 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`.

setdefwinpos(h, v) Sets the default window position.

setdefwinsize(width, height) Sets the default window size.

menucreate(title) Creates a menu object referring to a global menu (a menu that appears in all windows). Methods of menu objects are described below.

fleep() Causes a beep or bell (or perhaps a 'visual bell' or flash, hence the name).

message(string) Displays a dialog box containing the string. The user must click OK before the function returns.

askync(prompt, default) Displays a dialog that prompts the user to answer a question with yes or no. The function returns 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) Displays 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) Asks 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) Stores 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) Returns the contents of the system's cut buffer number `i`.

rotatebutbuffers(n) On X11, this rotates the 8 cut buffers by `n`. Ignored on the Macintosh.

getselection(i) Returns 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) Resets 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).[5]

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.

[5]There is no way yet to set the current font. This will change in a future version.

3.6.2 Window Object Methods

Window objects are created by `stdwin.open()`. There is no explicit function to close a window; windows are closed when they are garbage-collected. Window objects have the following methods:

begindrawing() Returns a drawing object, whose methods (described below) allow drawing in the window.

change(rect) Invalidates the given rectangle; this may cause a draw event.

gettitle() Returns the window's title string.

getdocsize() Returns a pair of integers giving the size of the document as set by `setdocsize()`.

getorigin() Returns a pair of integers giving the origin of the window with respect to the document.

getwinsize() Returns a pair of integers giving the size of the window.

menucreate(title) Creates a menu object referring to a local menu (a menu that appears only in this window). Methods menu objects are described below.

scroll(rect, point) Scrolls the given rectangle by the vector given by the point.

setwincursor(name) Sets 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 are `'ibeam'`, `'arrow'`, `'cross'`, `'watch'` and `'plus'`. On X11, there are many more (see `<X11/cursorfont.h>`).

setdocsize(point) Sets the size of the drawing document.

setorigin(point) Moves the origin of the window to the given point in the document.

setselection(i, str) Attempts to set X11 selection number `i` to the string `str`. (See stdwin method `getselection()` for the meaning of `i`.) Returns true if it succeeds. If it 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.

settitle(title) Sets the window's title string.

settimer(dsecs) Schedules a timer event for the window in `dsecs/10` seconds.

show(rect) Tries to ensure that the given rectangle of the document is visible in the window.

textcreate(rect) Creates a text-edit object in the document at the given rectangle. Methods of text-edit objects are described below.

3.6.3 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) Draws a box around a rectangle.

circle(center, radius) Draws a circle with given center point and radius.

elarc(center, (rh, rv), (a1, a2)) Draws 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) Erases a rectangle.

invert(rect) Inverts a rectangle.

line(p1, p2) Draws a line from point `p1` to `p2`.

paint(rect) Fills a rectangle.

text(p, str) Draws a string starting at point p (the point specifies the top left coordinate of the string).

shade(rect, percent) Fills a rectangle with a shading pattern that is about `percent` percent filled.

xorline(p1, p2) Draws a line in XOR mode.

baseline(), lineheight(), textbreak(), textwidth() These functions are similar to the corresponding functions described above for the `stdwin` module, but use the current font of the window instead of the (global) default font.

3.6.4 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) Adds a menu item with given text. The shortcut must be a string of length 1, or omitted (to specify no shortcut).

setitem(i, text) Sets the text of item number `i`.

enable(i, flag) Enables or disables item `i`.

check(i, flag) Sets or clears the *check mark* for item `i`.

3.6.5 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) Passes 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) Passes a draw event to the text-edit block. The rectangle specifies the redraw area.

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

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

getfocustext() Returns the text in the focus.

getrect() Returns 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() Returns the entire text buffer.

move(rect) Specifies a new position for the text-edit block in the document.

replace(str) Replaces the focus by the given string. The new focus is an insert point at the end of the string.

setfocus(i, j) Specifies the new focus. Out-of-bounds values are silently clipped.

3.6.6 Example

Here is a simple 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()
```

3.7 Built-in Module `amoeba`

This module provides some object types and operations useful for Amoeba applications. It is only available on systems that support Amoeba operations. RPC errors and other Amoeba errors are reported as the exception `amoeba.error = 'amoeba.error'`. The module `amoeba` defines the following items:

name_append(path, cap) Stores a capability in the Amoeba directory tree. Arguments are the pathname (a string) and the capability (a capability object as returned by `name_lookup()`).

name_delete(path) Deletes a capability from the Amoeba directory tree. Argument is the pathname.

name_lookup(path) Looks up a capability. Argument is the pathname. Returns a *capability* object, to which various interesting operations apply, described below.

name_replace(path, cap) Replaces a capability in the Amoeba directory tree. Arguments are the pathname and the new capability. (This differs from `name_append()` in the behavior when the pathname already exists: `name_append()` finds this an error while `name_replace()` allows it, as its name suggests.)

capv(A) table representing the capability environment at the time the interpreter was started. (Alas, modifying this table does not affect the capability environment of the interpreter.) For example, `amoeba.capv['ROOT']` is the capability of your root directory, similar to `getcap("ROOT")` in C.

error = `'amoeba.error'`
The exception raised when an Amoeba function returns an error. The value accompanying this exception is a pair containing the numeric error code and the corresponding string, as returned by the C function `err_why()`.

timeout(msecs) Sets the transaction timeout, in milliseconds. Returns the previous timeout. Initially, the timeout is set to 2 seconds by the Python interpreter.

3.7.1 Capability Operations

Capabilities are written in a convenient ASCII format, also used by the Amoeba utilities *c2a*(U) and *a2c*(U). For example:

```
>>> amoeba.name_lookup('/profile/cap')
aa:1c:95:52:6a:fa/14(ff)/8e:ba:5b:8:11:1a
>>>
```

The following methods are defined for capability objects.

dir_list() Returns a list of the names of the entries in an Amoeba directory.

b_read(offset, maxsize) Reads (at most) `maxsize` bytes from a bullet file at offset `offset`. The data is returned as a string. EOF is reported as an empty string.

b_size() Returns the size of a bullet file.

dir_append(), dir_delete(), dir_lookup(), dir_replace()
Like the corresponding `name_*` functions, but with a path relative to the capability. (For paths beginning with a slash the capability is ignored, since this is the defined semantics for Amoeba.)

std_info() Returns the standard info string of the object.

tod_gettime() Returns the time (in seconds since the Epoch, in UCT, as for POSIX) from a time server.

tod_settime(t) Sets the time kept by a time server.

3.8 Built-in Module `audio`

This module provides rudimentary access to the audio I/O device `/dev/audio` on the Silicon Graphics Personal IRIS; see audio(7). It supports the following operations:

setoutgain(n) Sets the output gain (0-255).

getoutgain() Returns the output gain.

setrate(n) Sets the sampling rate: 1=32K/sec, 2=16K/sec, 3=8K/sec.

setduration(n) Sets the 'sound duration' in units of 1/100 seconds.

read(n) Reads a chunk of `n` sampled bytes from the audio input (line in or microphone). The chunk is returned as a string of length n. Each byte encodes one sample as a signed 8-bit quantity using linear encoding. This string can be converted to numbers using `chr2num()` described below.

write(buf) Writes a chunk of samples to the audio output (speaker).

These operations support asynchronous audio I/O:

start_recording(n) Starts a second thread (a process with shared memory) that begins reading `n` bytes from the audio device. The main thread immediately continues.

wait_recording() Waits for the second thread to finish and returns the data read.

stop_recording() Makes the second thread stop reading as soon as possible. Returns the data read so far.

poll_recording() Returns true if the second thread has finished reading (so `wait_recording()` would return the data without delay).

`start_playing(chunk)`, `wait_playing()`, `stop_playing()`, `poll_playing()` Similar but for output. `stop_playing()` returns a lower bound for the number of bytes actually played (not very accurate).

The following operations do not affect the audio device but are implemented in C for efficiency:

amplify(buf, f1, f2) Amplifies a chunk of samples by a variable factor changing from `f1`/256 to `f2`/256. Negative factors are allowed. Resulting values that are to large to fit in a byte are clipped.

reverse(buf) Returns a chunk of samples backwards.

add(buf1, buf2) Bytewise adds two chunks of samples. Bytes that exceed the range are clipped. If one buffer shorter, it is assumed to be padded with zeros.

chr2num(buf) Converts a string of sampled bytes as returned by `read()` into a list containing the numeric values of the samples.

num2chr(list) Converts a list as returned by `chr2num()` back to a buffer acceptable by `write()`.

3.9 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() 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()
```

3.10 Built-in Module `pnl`

This module provides access to the *Panel Library* built by NASA Ames (to get it, send e-mail to `panel-requ...@nas.nasa.gov`). All access to it should be done through the standard module `panel`, which transparantly exports most functions from `pnl` but redefines `pnl.dopanel()`.

Warning: the Python interpreter will dump core if you don't create a GL window before calling `pnl.mkpanel()`.

The module is too large to document here in its entirety.

4 Standard Modules

The following standard modules are defined. They are available in one of the directories in the default module search path (try printing `sys.path` to find out the default search path.)

4.1 Standard Module `string`

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

digits(T) he string `'0123456789'`.

hexdigits(T) he string `'0123456789abcdefABCDEF'`.

letters(T) he concatenation of the strings `lowercase` and `uppercase` described below.

lowercase(T) he string `'abcdefghijklmnopqrstuvwxyz'`.

octdigits(T) he string `'01234567'`.

uppercase(T) he string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

whitespace(A) string containing all characters that are considered whitespace, i.e., space, tab and newline. This definition is used by `split()` and `strip()`.

The exceptions are:

atoi_error = `'non-numeric argument to string.atoi'`
Exception raised by `atoi` when a non-numeric string argument is detected. The exception argument is the offending string.

index_error = `'substring not found in string.index'`
Exception raised by `index` when `sub` is not found. The argument are the offending arguments to index: `(s, sub)`.

The functions are:

atoi(s) Converts a string to a number. The string must consist of one or more digits, optionally preceded by a sign (`'+'` or `'-'`).

index(s, sub) Returns the lowest index in `s` where the substring `sub` is 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.

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.

4.2 Standard Module `path`

This module implements some useful functions on POSIX pathnames.

basename(p) Returns the base name of pathname `p`. This is the second half of the pair returned by `path.split(p)`.

cat(p, q) Performs intelligent pathname concatenation on paths `p` and `q`: 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.

commonprefix(list) Returns the longest string that is a prefix of all strings in `list`. If `list` is empty, the empty string (`''`) is returned.

exists(p) Returns true if `p` refers to an existing path.

isdir(p) Returns true if `p` refers to an existing directory.

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

ismount(p) Returns true if `p` is an absolute path that occurs in the mount table as output by the `/etc/mount` utility. This output is read once when the function is used for the first time.[6]

split(p) Returns a pair `(head, tail)` such that `tail` contains no slashes and `path.cat(head, tail)` is equal to `p`.

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.)

4.3 Standard Module `getopt`

This module helps scripts to parse the command line arguments in `sys.argv`. It uses the same conventions as the UNIX `getopt()` function. 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

[6] Is there a better way to check for mount points?

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']
>>>
```

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.

4.4 Standard Module `rand`

This module implements a pseudo-random number generator 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.

4.5 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.

4.6 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 *
>>>
```

4.7 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 = `'rect.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(T) he 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* $\leq$ *bottom*.

intersect(list) Returns the intersection of all rectangles in the list argument. It may also be called with a tuple argument or with two or more rectangles as arguments. 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)*.

4.8 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 files for details.

4.9 Standard Module `panel`

This module should be used instead of the built-in module `pnl` to interface with the *Panel Library*.

The module is too large to document here in its entirety. One interesting function:

defpanellist(filename) Parses a panel description file containing S-expressions written by the *Panel Editor* that accompanies the Panel Library and creates the described panels. It returns a list of panel objects.

Warning: the Python interpreter will dump core if you don't create a GL window before calling `panel.mkpanel()` or `panel.defpanellist()`.

4.10 Standard Module `panelparser`

This module defines a self-contained parser for S-expressions as output by the Panel Editor (which is written in Scheme so it can't help writing S-expressions). The relevant function is `panelparser.parse_file(file)` which has a file object (not a filename!) as argument and returns a list of parsed S-expressions. Each S-expression is converted into a Python list, with atoms converted to Python strings and sub-expressions (recursively) to Python lists. For more details, read the module file.

5 P.M.

commands

cmp?

*cache?

localtime?

calendar?

__dict?

mac?