Internal Objects -- Python library reference

2.1.7.9. Internal Objects

(See the Python Reference Manual for these.)

gdbm -- Python library reference

8.6. Built-in Module gdbm

This module is nearly identical to the `dbm` module, but uses GDBM instead. Its interface is identical, and not repeated here.

Warning: the file formats created by gdbm and dbm are incompatible.

posixfile -- Python library reference

8.10. Standard Module posixfile

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

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

The posixfile module defines the following constants:

SEEK_SET -- data of module posixfile

> offset is calculated from the start of the file

SEEK_CUR -- data of module posixfile

> offset is calculated from the current position in the file

SEEK_END -- data of module posixfile

> offset is calculated from the end of the file

The posixfile module defines the following functions:

open *(filename[, mode[, bufsize]])* -- function of module posixfile

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

fileopen *(fileobject)* -- function of module posixfile

> Create a new posixfile object with the given standard file object. The resulting object has the same filename and mode as the original file object.

The posixfile object defines the following additional methods:

lock *(fmt, [len[, start [, whence]]])* -- Method on posixfile

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

flags *([flags])* -- Method on posixfile

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

dup *()* -- Method on posixfile

> Duplicate the file object and the underlying file pointer and file descriptor. The resulting object behaves as if it were newly opened.

dup2 *(fd)* -- Method on posixfile

Duplicate the file object and the underlying file pointer and file descriptor. The new object will have the given file descriptor. Otherwise the resulting object behaves as if it were newly opened.

file *()* -- Method on posixfile

Return the standard file object that the posixfile object is based on. This is sometimes necessary for functions that insist on a standard file object.

All methods return `IOError` when the request fails.

Format characters for the `lock()` method have the following meaning:

Format

Meaning ---

`` `u' ``

unlock the specified region

`` `r' ``

request a read lock for the specified section

`` `w' ``

request a write lock for the specified section

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

Modifier

Meaning --- *Notes*

`` `|' ``

wait until the lock has been granted

`` `?' ``

return the first lock conflicting with the requested lock, or `None` if there is no conflict. --- (1)

Note:

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

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

Format

Meaning ---

`` `a' ``

append only flag

`` `c' ``

close on exec flag

`` `n' ``

no delay flag (also called non-blocking flag)

`` `s' ``

synchronization flag

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

Modifier

Meaning --- *Notes*

`` `!' ``

turn the specified flags 'off', instead of the default 'on' --- (1)

`` `=' ``

replace the flags, instead of the default 'OR' operation --- (1)

`` `?' ``

return a string in which the characters represent the flags that are set. --- (2)

Note:

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

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

Examples:

```
from posixfile import *
```

file = open('/tmp/test', 'w')

file.lock('w|')

...

file.lock('u')

file.close()

Classes and Instances -- Python library reference

2.1.7.2. Classes and Class Instances

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

socket -- Python library reference

Next: select Prev: signal Up: Optional Operating System Services Top: Top

7.2. Built-in Module socket

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

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

The Python interface is a straightforward transliteration of the UNIX system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a socket object whose methods implement the various socket system calls. Parameter types are somewhat 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 mode is supported through the `setblocking()` method.

The module `socket` exports the following constants and functions:

error -- exception of module socket

> 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 -- data of module socket

AF_INET -- data of module socket

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

SOCK_STREAM -- data of module socket

SOCK_DGRAM -- data of module socket

SOCK_RAW -- data of module socket

SOCK_RDM -- data of module socket

SOCK_SEQPACKET -- data of module socket

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

SO_* -- data of module socket

SOMAXCONN -- data of module socket

MSG_* -- data of module socket

SOL_* -- data of module socket

IPPROTO_* -- data of module socket

IPPORT_* -- data of module socket

INADDR_* -- data of module socket

IP_* -- data of module socket

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

gethostbyname *(hostname)* -- function of module socket

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

gethostname *()* -- function of module socket

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

gethostbyaddr *(ip_address)* -- function of module socket

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

getservbyname *(servicename, protocolname)* -- function of module socket

> 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])* -- function of module socket

> 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])* -- function of module socket

> 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).

Menu

Socket Objects

Socket Example

speech channel objects -- Python library reference

Prev: voice objects Up: macspeech Top: Top

14.10.2. speech channel objects

A speech channel object allows you to speak strings with slightly more control than `SpeakString()`, and allows you to use multiple speakers at the same time. Please note that channel pitch and rate are interrelated in some way, so that to make your Macintosh sing you will have to adjust both.

SpeakText *(str)* -- Method on speech channel object

Start uttering the given string.

Stop *()* -- Method on speech channel object

Stop babbling.

GetPitch *()* -- Method on speech channel object

Return the current pitch of the channel, as a floating-point number.

SetPitch *(pitch)* -- Method on speech channel object

Set the pitch of the channel.

GetRate *()* -- Method on speech channel object

Get the speech rate (utterances per minute) of the channel as a floating point number.

SetRate *(rate)* -- Method on speech channel object

Set the speech rate of the channel.

Prev: voice objects Up: macspeech Top: Top

macconsole options object -- Python library reference

14.4.1. macconsole options object

These options are examined when a window is created:

top -- option of macconsole

left -- option of macconsole

> The origin of the window.

nrows -- option of macconsole

ncols -- option of macconsole

> The size of the window.

txFont -- option of macconsole

txSize -- option of macconsole

txStyle -- option of macconsole

> The font, fontsize and fontstyle to be used in the window.

title -- option of macconsole

> The title of the window.

pause_atexit -- option of macconsole

> If set non-zero, the window will wait for user action before closing.

httplib -- Python library reference

11.3. Standard Module httplib

This module defines a class which implements the client side of the HTTP protocol. It is normally not used directly --- the module `urllib` uses it to handle URLs that use HTTP. The module defines one class, `HTTP`. An `HTTP` instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If no host is passed, no connection is made, and the `connect` method should be used to connect to a server. For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = httplib.HTTP('www.cwi.nl')
```

>>> h2 = httplib.HTTP('www.cwi.nl:80')

>>> h3 = httplib.HTTP('www.cwi.nl', 80)

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

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

Menu

HotProfile Class -- Python library reference

10.8.2. HotProfile Class

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

```
class HotProfile(Profile):

    def trace_dispatch_exception(self, frame, t):

        rt, rtt, rfn, rframe, rcur = self.cur

        if rcur and not rframe is frame:

            return self.trace_dispatch_return(rframe, t)

        return 0

    def trace_dispatch_call(self, frame, t):

        self.cur = (t, 0, frame, self.cur)

        return 1

    def trace_dispatch_return(self, frame, t):

        rt, rtt, frame, rcur = self.cur
```

```
        rfn = `frame.f_code`

        pt, ptt, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pframe, pcur

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

        return 1


    def snapshot_stats(self):
        self.stats = {}
        for func in self.timings.keys():
            nc, tt = self.timings[func]
            nor_func = self.func_normalize(func)
```

```
self.stats[nor_func] = nc, nc, tt, 0, {}
```

Prev: OldProfile Class Up: Profiler Extensions Top: Top

Next: macconsole Prev: macpath Up: Macintosh Specific Services Top: Top

14.3. Built-in Module ctb

This module provides a partial interface to the Macintosh Communications Toolbox. Currently, only Connection Manager tools are supported. It may not be available in all Mac Python versions.

error -- data of module ctb

The exception raised on errors.

cmData -- data of module ctb

cmCntl -- data of module ctb

cmAttn -- data of module ctb

Flags for the *channel* argument of the *Read* and *Write* methods.

cmFlagsEOM -- data of module ctb

End-of-message flag for *Read* and *Write*.

choose* -- data of module ctb

Values returned by *Choose*.

cmStatus* -- data of module ctb

Bits in the status as returned by *Status*.

available *()* -- function of module ctb

Return 1 if the communication toolbox is available, zero otherwise.

CMNew *(name, sizes)* -- function of module ctb

Create a connection object using the connection tool named *name*. *sizes* is a 6-tuple given buffer sizes for data in, data out, control in, control out, attention in and attention out. Alternatively, passing `None` will result in default buffer sizes.

Menu

Next: macconsole Prev: macpath Up: Macintosh Specific Services Top: Top

stdwin -- Python library reference

15.1. Built-in Module stdwin

This module defines several new object types and functions that provide access to the functionality of STDWIN.

On Unix running X11, it can only be used if the `DISPLAY` environment variable is set or an explicit `-display *displayname*' argument is passed to the Python interpreter.

Functions have names that usually resemble their C STDWIN counterparts with the initial `w' dropped. Points are represented by pairs of integers; rectangles by pairs of points. For a complete description of STDWIN please refer to the documentation of STDWIN for C programmers (aforementioned CWI report).

Menu

Module Index -- Python library reference

Module Index

Next: Concept Index Prev: Variable Index Up: Top Top: Top

mac -- Python library reference

14.1. Built-in Module mac

This module provides a subset of the operating system dependent functionality provided by the optional built-in module `posix`. It is best accessed through the more portable standard module `os`.

The following functions are available in this module: `chdir`, `getcwd`, `listdir`, `mkdir`, `rename`, `rmdir`, `stat`, `sync`, `unlink`, as well as the exception `error`.

Special Attributes -- Python library reference

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__` yields `['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.

macconsole -- Python library reference

14.4. Built-in Module macconsole

This module is available on the Macintosh, provided Python has been built using the Think C compiler. It provides an interface to the Think console package, with which basic text windows can be created.

options -- data of module macconsole

An object allowing you to set various options when creating windows, see below.

C_ECHO -- data of module macconsole

C_NOECHO -- data of module macconsole

C_CBREAK -- data of module macconsole

C_RAW -- data of module macconsole

Options for the `setmode` method. *C_ECHO* and *C_CBREAK* enable character echo, the other two disable it, *C_ECHO* and *C_NOECHO* enable line-oriented input (erase/kill processing, etc).

copen *()* -- function of module macconsole

Open a new console window. Return a console window object.

fopen *(fp)* -- function of module macconsole

Return the console window object corresponding with the given file object. *fp* should be one of `sys.stdin`, `sys.stdout` or `sys.stderr`.

Menu

macconsole options object

console window object

rand -- Python library reference

5.2. Standard Module rand

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

rand *()* -- function of module rand

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

choice *(s)* -- function of module rand

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

srand *(seed)* -- function of module rand

> 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.

types -- Python library reference

3.2. Standard Module types

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

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

```
from types import *

def delete(list, item):

    if type(item) is IntType:

        del list[item]

    else:

        list.remove(item)
```

The module defines the following names:

NoneType -- data of module types

The type of `None`.

TypeType -- data of module types

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

IntType -- data of module types

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

LongType -- data of module types

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

FloatType -- data of module types

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

StringType -- data of module types

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

TupleType -- data of module types

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

ListType -- data of module types

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

DictType -- data of module types

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

DictionaryType -- data of module types

An alternative name for `DictType`.

FunctionType -- data of module types

The type of user-defined functions and lambdas.

LambdaType -- data of module types

An alternative name for `FunctionType`.

CodeType -- data of module types

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

ClassType -- data of module types

The type of user-defined classes.

InstanceType -- data of module types

The type of instances of user-defined classes.

MethodType -- data of module types

The type of methods of user-defined class instances.

UnboundMethodType -- data of module types

An alternative name for `MethodType`.

BuiltinFunctionType -- data of module types

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

BuiltinMethodType -- data of module types

An alternative name for `BuiltinFunction`.

ModuleType -- data of module types

The type of modules.

FileType -- data of module types

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

XRangeType -- data of module types

The type of range objects returned by `xrange()`.

TracebackType -- data of module types

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

FrameType -- data of module types

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

Optional Operating System Services -- Python library reference

7. Optional Operating System Services

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

signal

--- Set handlers for asynchronous events.

socket

--- Low-level networking interface.

select

--- Wait for I/O completion on multiple streams.

thread

--- Create multiple threads of control within one namespace.

Menu

binhex -- Python library reference

11.12. Standard module binhex

This module encodes and decodes files in binhex4 format, a format allowing representation of Macintosh files in ASCII. On the macintosh, both forks of a file and the finder information are encoded (or decoded), on other platforms only the data fork is handled.

The `binhex` module defines the following functions:

binhex *(input, output)* -- function of module binhex

> Convert a binary file with filename *input* to binhex file *output*. The *output* parameter can either be a filename or a file-like object (any object supporting a *write* and *close* method).

hexbin *(input[, output])* -- function of module binhex

> Decode a binhex file *input*. *Input* may be a filename or a file-like object supporting *read* and *close* methods. The resulting file is written to a file named *output*, unless the argument is empty in which case the output filename is read from the binhex file.

Menu

notes

SGI IRIX Specific Services -- Python library reference

16. SGI IRIX Specific Services

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

Menu

dnr result object -- Python library reference

14.5.1. dnr result object

Since the DNR calls all execute asynchronously you do not get the results back immediately. Instead, you get a dnr result object. You can check this object to see whether the query is complete, and access its attributes to obtain the information when it is.

Alternatively, you can also reference the result attributes directly, this will result in an implicit wait for the query to complete.

The *rtnCode* and *cname* attributes are always available, the others depend on the type of query (address, hinfo or mx).

wait *()* -- Method on dnr result object

> Wait for the query to complete.

isdone *()* -- Method on dnr result object

> Return 1 if the query is complete.

rtnCode -- attribute of dnr result object

> The error code returned by the query.

cname -- attribute of dnr result object

> The canonical name of the host that was queried.

ip0 -- attribute of dnr result object

ip1 -- attribute of dnr result object

ip2 -- attribute of dnr result object

ip3 -- attribute of dnr result object

> At most four integer IP addresses for this host. Unused entries are zero. Valid only for address queries.

cpuType -- attribute of dnr result object

osType -- attribute of dnr result object

> Textual strings giving the machine type an OS name. Valid for hinfo queries.

exchange -- attribute of dnr result object

> The name of a mail-exchanger host. Valid for mx queries.

preference -- attribute of dnr result object

> The preference of this mx record. Not too useful, since the Macintosh will only return a single mx record. Mx queries only.

The simplest way to use the module to convert names to dotted-decimal strings, without worrying about idle time, etc:

```
>>> def gethostname(name):
```

...   import macdnr

```
...     dnrr = macdnr.StrToAddr(name)

...     return macdnr.AddrToStr(dnrr.ip0)
```

Instant Users Manual -- Python library reference

10.3. Instant Users Manual

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

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

```
import profile


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

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

```
import profile


profile.run("foo()", 'fooprof')
```

When you wish to review the profile, you should use the methods in the `pstats` module. Typically you would load the statistics data as follows:

```
import pstats


p = pstats.Stats('fooprof')
```

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

```
p.strip_dirs().sort_stats(-1).print_stats()
```

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

```
p.sort_stats('name')
```

p.print_stats()

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

```
p.sort_stats('cumulative').print_stats(10)
```

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

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

```
p.sort_stats('time').print_stats(10)
```

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

You might also try:

```
p.sort_stats('file').print_stats('__init__')
```

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

```
p.sort_stats('time', 'cum').print_stats(.5, 'init')
```

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

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

```
p.print_callers(.5, 'init')
```

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

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

```
p.print_callees()
```

p.add('fooprof')

aifc -- Python library reference

12.3. Standard Module aifc

This module provides support for reading and writing AIFF and AIFF-C files. AIFF is Audio Interchange File Format, a format for storing digital audio samples in a file. AIFF-C is a newer version of the format that includes the ability to compress the audio data.

Audio files have a number of parameters that describe the audio data. The sampling rate or frame rate is the number of times per second the sound is sampled. The number of channels indicate if the audio is mono, stereo, or quadro. Each frame consists of one sample per channel. The sample size is the size in bytes of each sample. Thus a frame consists of *nchannels***samplesize* bytes, and a second's worth of audio consists of *nchannels***samplesize***framerate* bytes.

For example, CD quality audio has a sample size of two bytes (16 bits), uses two channels (stereo) and has a frame rate of 44,100 frames/second. This gives a frame size of 4 bytes (2*2), and a second's worth occupies 2*2*44100 bytes, i.e. 176,400 bytes.

Module `aifc` defines the following function:

open *(file, mode)* -- function of module aifc

> Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument file is either a string naming a file or a file object. The mode is either the string `'r'` when the file must be opened for reading, or `'w'` when the file must be opened for writing. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`.

Objects returned by `aifc.open()` when a file is opened for reading have the following methods:

getnchannels *()* -- Method on aifc object

> Return the number of audio channels (1 for mono, 2 for stereo).

getsampwidth *()* -- Method on aifc object

> Return the size in bytes of individual samples.

getframerate *()* -- Method on aifc object

> Return the sampling rate (number of audio frames per second).

getnframes *()* -- Method on aifc object

> Return the number of audio frames in the file.

getcomptype *()* -- Method on aifc object

> Return a four-character string describing the type of compression used in the audio file. For AIFF files, the returned value is `'NONE'`.

getcompname *()* -- Method on aifc object

> Return a human-readable description of the type of compression used in the audio file. For AIFF files, the returned value is `'not compressed'`.

getparams *()* -- Method on aifc object

> Return a tuple consisting of all of the above values in the above order.

getmarkers *()* -- Method on aifc object

> Return a list of markers in the audio file. A marker consists of a tuple of three elements. The first is

the mark ID (an integer), the second is the mark position in frames from the beginning of the data (an integer), the third is the name of the mark (a string).

getmark *(id)* -- Method on aifc object

Return the tuple as described in `getmarkers` for the mark with the given id.

readframes *(nframes)* -- Method on aifc object

Read and return the next *nframes* frames from the audio file. The returned data is a string containing for each frame the uncompressed samples of all channels.

rewind *()* -- Method on aifc object

Rewind the read pointer. The next `readframes` will start from the beginning.

setpos *(pos)* -- Method on aifc object

Seek to the specified frame number.

tell *()* -- Method on aifc object

Return the current frame number.

close *()* -- Method on aifc object

Close the AIFF file. After calling this method, the object can no longer be used.

Objects returned by `aifc.open()` when a file is opened for writing have all the above methods, except for `readframes` and `setpos`. In addition the following methods exist. The `get` methods can only be called after the corresponding `set` methods have been called. Before the first `writeframes` or `writeframesraw`, all parameters except for the number of frames must be filled in.

aiff *()* -- Method on aifc object

Create an AIFF file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

aifc *()* -- Method on aifc object

Create an AIFF-C file. The default is that an AIFF-C file is created, unless the name of the file ends in '.aiff' in which case the default is an AIFF file.

setnchannels *(nchannels)* -- Method on aifc object

Specify the number of channels in the audio file.

setsampwidth *(width)* -- Method on aifc object

Specify the size in bytes of audio samples.

setframerate *(rate)* -- Method on aifc object

Specify the sampling frequency in frames per second.

setnframes *(nframes)* -- Method on aifc object

Specify the number of frames that are to be written to the audio file. If this parameter is not set, or not set correctly, the file needs to support seeking.

setcomptype *(type, name)* -- Method on aifc object

Specify the compression type. If not specified, the audio data will not be compressed. In AIFF files, compression is not possible. The name parameter should be a human-readable description of the compression type, the type parameter should be a four-character string. Currently the following compression types are supported: NONE, ULAW, ALAW, G722.

setparams *(nchannels, sampwidth, framerate, comptype, compname)* -- Method on aifc object

Set all the above parameters at once. The argument is a tuple consisting of the various parameters. This means that it is possible to use the result of a `getparams` call as argument to `setparams`.

setmark *(id, pos, name)* -- Method on aifc object

Add a mark with the given id (larger than 0), and the given name at the given position. This method can be called at any time before `close`.

tell *()* -- Method on aifc object

Return the current write position in the output file. Useful in combination with `setmark`.

writeframes *(data)* -- Method on aifc object

Write data to the output file. This method can only be called after the audio file parameters have been set.

writeframesraw *(data)* -- Method on aifc object

Like `writeframes`, except that the header of the audio file is not updated.

close *()* -- Method on aifc object

Close the AIFF file. The header of the file is updated to reflect the actual size of the audio data. After calling this method, the object can no longer be used.

HTTP Objects -- Python library reference

11.3.1. HTTP Objects

`HTTP` instances have the following methods:

set_debuglevel *(level)* -- Method on HTTP

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

connect *(host[, port])* -- Method on HTTP

> Connect to the server given by *host* and *port*. See the intro for the default port. This should be called directly only if the instance was instantiated without passing a host.

send *(data)* -- Method on HTTP

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

putrequest *(request, selector)* -- Method on HTTP

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

putheader *(header, argument[, ...])* -- Method on HTTP

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

endheaders *()* -- Method on HTTP

> Send a blank line to the server, signalling the end of the headers.

getreply *()* -- Method on HTTP

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

getfile *()* -- Method on HTTP

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

array -- Python library reference

5.4. Built-in Module array

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

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 speaking, by the C implementation). The actual size can be accessed through the *itemsize* attribute.

See also built-in module `struct`. The module defines the following function:

array *(typecode[, initializer])* -- function of module array

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 -- data of module array

The typecode character used to create the array.

itemsize -- data of module array

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

append *(x)* -- function of module array

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

byteswap *(x)* -- function of module array

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

fromfile *(f, n)* -- function of module array

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

fromlist *(list)* -- function of module array

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

fromstring *(s)* -- function of module array

> 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)* -- function of module array

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

tofile *(f)* -- function of module array

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

tolist *()* -- function of module array

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

tostring *()* -- function of module array

> 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])

fm -- Python library reference

16.7. Built-in Module fm

This module provides access to the IRIS *Font Manager* library. It is available only on Silicon Graphics machines. See also: 4Sight User's Guide, Section 1, Chapter 5: Using the IRIS Font Manager.

This is not yet a full interface to the IRIS Font Manager. Among the unsupported features are: matrix operations; cache operations; character operations (use string operations instead); some details of font info; individual glyph metrics; and printer matching.

It supports the following operations:

init *()* -- function of module fm

> 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)* -- function of module fm

> Return a font handle object. Calls `fmfindfont(`*`fontname)`*.

enumerate *()* -- function of module fm

> Returns a list of available font names. This is an interface to `fmenumerate()`.

prstr *(string)* -- function of module fm

> Render a string using the current font (see the `setfont()` font handle method below). Calls `fmprstr(`*`string)`*.

setpath *(string)* -- function of module fm

> Sets the font search path. Calls `fmsetpath(string)`. (XXX Does not work!?!)

fontpath *()* -- function of module fm

> Returns the current font search path.

Font handle objects support the following operations:

scalefont *(factor)* -- Method on font handle

> Returns a handle for a scaled version of this font. Calls `fmscalefont(`*`fh, factor)`*.

setfont *()* -- Method on font handle

> Makes this font the current font. Note: the effect is undone silently when the font handle object is deleted. Calls `fmsetfont(`*`fh)`*.

getfontname *()* -- Method on font handle

> Returns this font's name. Calls `fmgetfontname(`*`fh)`*.

getcomment *()* -- Method on font handle

> Returns the comment string associated with this font. Raises an exception if there is none. Calls `fmgetcomment(`*`fh)`*.

getfontinfo *()* -- Method on font handle

> 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)* -- Method on font handle

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

Next: gl Prev: flp Up: SGI IRIX Specific Services Top: Top

gopherlib -- Python library reference

11.5. Standard Module gopherlib

This module provides a minimal implementation of client side of the the Gopher protocol. It is used by the module `urllib` to handle URLs that use the Gopher protocol.

The module defines the following functions:

send_selector *(selector, host[, port])* -- function of module gopherlib

> Send a *selector* string to the gopher server at *host* and *port* (default 70). Return an open file object from which the returned document can be read.

send_query *(selector, query, host[, port])* -- function of module gopherlib

> Send a *selector* string and a *query* string to a gopher server at *host* and *port* (default 70). Return an open file object from which the returned document can be read.

Note that the data returned by the Gopher server can be of any type, depending on the first character of the selector string. If the data is text (first character of the selector is \``0`'), lines are terminated by CRLF, and the data is terminated by a line consisting of a single \``.`', and a leading \``.`' should be stripped from lines that begin with \``..`'. Directory listings (first charactger of the selector is \``1`') are transferred using the same protocol.

Concept Index -- Python library reference

Concept Index

Table of Contents

Prev: Module Index Up: Top Top: Top

Truth Value Testing -- Python library reference

2.1.1. Truth Value Testing

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

`None`

zero of any numeric type, e.g., `0`, `0L`, `0.0`.

any empty sequence, e.g., `''`, `()`, `[]`.

any empty mapping, e.g., `{}`.

instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns zero.

All other values are considered true --- so objects of many types are always true. Operations and built-in functions that have a Boolean result always return `0` for false and `1` for true, unless otherwise stated. (Important exception: the Boolean operations \``or`' and \``and`' always return one of their operands.)

DialogWindow Objects -- Python library reference

14.12.3. DialogWindow Objects

DialogWindow objects have the following methods besides those of `Window` objects:

open *(resid)* -- Method on DialogWindow

> Create the dialog window, from the DLOG resource with id *resid*. The dialog object is stored in `self.wid`.

do_itemhit *(item, event)* -- Method on DialogWindow

> Item number *item* was hit. You are responsible for redrawing toggle buttons, etc.

FTP Objects -- Python library reference

11.4.1. FTP Objects

FTP instances have the following methods:

set_debuglevel *(level)* -- Method on FTP object

> Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

connect *(host[, port])* -- Method on FTP object

> Connect to the given host and port. The default port number is 21, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made.

getwelcome *()* -- Method on FTP object

> Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

login *([user[, passwd[, acct]]])* -- Method on FTP object

> Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to \``anonymous`'. If *user* is `anonymous`, the default *passwd* is \``realuser@host`' where *realuser* is the real user name (glanced from the \``LOGNAME`' or \``USER`' environment variable) and *host* is the hostname as returned by `socket.gethostname()`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in.

abort *()* -- Method on FTP object

> Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

sendcmd *(command)* -- Method on FTP object

> Send a simple command string to the server and return the response string.

voidcmd *(command)* -- Method on FTP object

> Send a simple command string to the server and handle the response. Return nothing if a response code in the range 200--299 is received. Raise an exception otherwise.

retrbinary *(command, callback, maxblocksize)* -- Method on FTP object

> Retrieve a file in binary transfer mode. *command* should be an appropriate \``RETR`' command, i.e. `"RETR filename"`. The *callback* function is called for each block of data received, with a single string argument giving the data block. The *maxblocksize* argument specifies the maximum block size (which may not be the actual size of the data blocks passed to *callback*).

retrlines *(command[, callback])* -- Method on FTP object

> Retrieve a file or directory listing in ASCII transfer mode. varcommand should be an appropriate \``RETR`' command (see `retrbinary()` or a \``LIST`' command (usually just the string `"LIST"`). The *callback* function is called for each line, with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.

storbinary *(command, file, blocksize)* -- Method on FTP object

Store a file in binary transfer mode. *command* should be an appropriate `` `STOR' `` command, i.e. `"STOR filename"`. *file* is an open file object which is read until EOF using its `read()` method in blocks of size *blocksize* to provide the data to be stored.

storlines *(command, file)* -- Method on FTP object

Store a file in ASCII transfer mode. *command* should be an appropriate `` `STOR' `` command (see `storbinary()`). Lines are read until EOF from the open file object *file* using its `readline()` method to privide the data to be stored.

nlst *(argument[, ...])* -- Method on FTP object

Return a list of files as returned by the `` `NLST' `` command. The optional varargument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `` `NLST' `` command.

dir *(argument[, ...])* -- Method on FTP object

Return a directory listing as returned by the `` `LIST' `` command, as a list of lines. The optional varargument is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `` `LIST' `` command. If the last argument is a function, it is used as a *callback* function as for `retrlines()`.

rename *(fromname, toname)* -- Method on FTP object

Rename file *fromname* on the server to *toname*.

cwd *(pathname)* -- Method on FTP object

Set the current directory on the server.

mkd *(pathname)* -- Method on FTP object

Create a new directory on the server.

pwd *()* -- Method on FTP object

Return the pathname of the current directory on the server.

quit *()* -- Method on FTP object

Send a `` `QUIT' `` command to the server and close the connection. This is the ``polite'' way to close a connection, but it may raise an exception of the server reponds with an error to the `QUIT` command.

close *()* -- Method on FTP object

Close the connection unilaterally. This should not be applied to an already closed connection (e.g. after a successful call to `quit()`.

FL Functions -- Python library reference

Next: Form Objects Prev: fl Up: fl Top: Top

16.4.1. Functions Defined in Module fl

Module `fl` defines the following functions. For more information about what they do, see the description of the equivalent C function in the FORMS documentation:

make_form *(type, width, height)* -- function of module fl

> Create a form with given type, width and height. This returns a form object, whose methods are described below.

do_forms *()* -- function of module fl

> The standard FORMS main loop. Returns a Python object representing the FORMS object needing interaction, or the special value `FL.EVENT`.

check_forms *()* -- function of module fl

> 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)* -- function of module fl

> Set the event callback function.

set_graphics_mode *(rgbmode, doublebuffering)* -- function of module fl

> Set the graphics modes.

get_rgbmode *()* -- function of module fl

> Return the current rgb mode. This is the value of the C global variable `fl_rgbmode`.

show_message *(str1, str2, str3)* -- function of module fl

> Show a dialog box with a three-line message and an OK button.

show_question *(str1, str2, str3)* -- function of module fl

> 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])* -- function of module fl

> Show a dialog box with a three-line message and up to three buttons. It returns the number of the button clicked by the user (`1`, `2` or `3`).

show_input *(prompt, default)* -- function of module fl

> 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)* -- function of module fl

> Show a dialog box in which the user can select a file. It returns the absolute filename selected by the user, or `None` if the user presses Cancel.

get_directory *()* -- function of module fl

get_pattern *()* -- function of module fl

get_filename *()* -- function of module fl

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)* -- function of module fl

unqdevice *(dev)* -- function of module fl

isqueued *(dev)* -- function of module fl

qtest *()* -- function of module fl

qread *()* -- function of module fl

qreset *()* -- function of module fl

qenter *(dev, val)* -- function of module fl

get_mouse *()* -- function of module fl

tie *(button, valuator1, valuator2)* -- function of module fl

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 *()* -- function of module fl

mapcolor *()* -- function of module fl

getmcolor *()* -- function of module fl

See the description in the FORMS documentation of `fl_color`, `fl_mapcolor` and `fl_getmcolor`.

EasyDialogs -- Python library reference

14.11. Standard module EasyDialogs

The `EasyDialogs` module contains some simple dialogs for the Macintosh, modelled after the `stdwin` dialogs with similar names.

The `EasyDialogs` module defines the following functions:

Message *(str)* -- function of module EasyDialogs

> A modal dialog with the message text *str*, which should be at most 255 characters long, is displayed. Control is returned when the user clicks ``OK''.

AskString *(prompt[, default])* -- function of module EasyDialogs

> Ask the user to input a string value, in a modal dialog. *Prompt* is the promt message, the optional *default* arg is the initial value for the string. All strings can be at most 255 bytes long. *AskString* returns the string entered or `None` in case the user cancelled.

AskYesNoCancel *(question[, default])* -- function of module EasyDialogs

> Present a dialog with text *question* and three buttons labelled ``yes'', ``no'' and ``cancel''. Return `1` for yes, `0` for no and `-1` for cancel. The default return value chosen by hitting return is `0`. This can be changed with the optional *default* argument.

Note that `EasyDialogs` does not currently use the notification manager. This means that displaying dialogs while the program is in the background will need to unexpected results and possibly crashes.

thread -- Python library reference

7.4. Built-in Module thread

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

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

It defines the following constant and functions:

error -- exception of module thread

> Raised on thread-specific errors.

start_new_thread *(func, arg)* -- function of module thread

> 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 terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

exit *()* -- function of module thread

> This is a shorthand for `thread.exit_thread()`.

exit_thread *()* -- function of module thread

> Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

allocate_lock *()* -- function of module thread

> Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

get_ident *()* -- function of module thread

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

Lock objects have the following methods:

acquire *([waitflag])* -- Method on lock

> 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 *()* -- Method on lock

> Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

locked *()* -- Method on lock

> 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. (When the `signal` module is available, interrupts always go to the main thread.)

Calling `sys.exit()` or raising the `SystemExit` is equivalent to calling `thread.exit_thread()`.

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

Prev: select Up: Optional Operating System Services Top: Top

rect -- Python library reference

15.3. Standard Module rect

This module contains useful operations on rectangles. A rectangle is defined as in module `stdwin`: a pair of points, where a point is a pair of integers. For example, the rectangle

```
(10, 20), (90, 80)
```

is a rectangle whose left, top, right and bottom edges are 10, 20, 90 and 80, respectively. Note that the positive vertical axis points down (as in `stdwin`).

The module defines the following objects:

error -- exception of module rect

> 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 -- data of module rect

> 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)* -- function of module rect

> Returns true if the given rectangle is empty. A rectangle `(left, top), (right, bottom)` is empty if `left >= right` or `top => bottom`.

intersect *(list)* -- function of module rect

> 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)* -- function of module rect

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)* -- function of module 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 <= h < right`* and *`top <= v < bottom`*.

inset *(rect, (dh, dv))* -- function of module rect

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)* -- function of module rect

Converts a rectangle to geometry representation: *`(left, top), (width, height)`*.

geom2rect *(geom)* -- function of module rect

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

Prev: stdwinevents Up: Standard Windowing Interface Top: Top

TCP Status Objects -- Python library reference

14.9.2. TCP Status Objects

This object has no methods, only some members holding information on the connection. A complete description of all fields in this objects can be found in the Apple documentation. The most interesting ones are:

localHost -- attribute of TCP status

localPort -- attribute of TCP status

remoteHost -- attribute of TCP status

remotePort -- attribute of TCP status

> The integer IP-addresses and port numbers of both endpoints of the connection.

sendWindow -- attribute of TCP status

> The current window size.

amtUnackedData -- attribute of TCP status

> The number of bytes sent but not yet acknowledged. `sendWindow - amtUnackedData` is what you can pass to `Send` without blocking.

amtUnreadData -- attribute of TCP status

> The number of bytes received but not yet read (what you can `Recv` without blocking).

Profiler Extensions -- Python library reference

10.8. Extensions --- Deriving Better Profilers

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

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

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

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

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

Menu

Port Objects -- Python library reference

Prev: Configuration Objects Up: al Top: Top

16.1.2. Port Objects

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

closeport *()* -- Method on audio port object

Close the port.

getfd *()* -- Method on audio port object

Return the file descriptor as an int.

getfilled *()* -- Method on audio port object

Return the number of filled samples.

getfillable *()* -- Method on audio port object

Return the number of fillable samples.

readsamps *(nsamples)* -- Method on audio port object

Read a number of samples from the queue, blocking if necessary. Return the data as a string containing the raw data, (e.g., 2 bytes per sample in big-endian byte order (high byte, low byte) if you have set the sample width to 2 bytes).

writesamps *(samples)* -- Method on audio port object

Write samples into the queue, blocking if necessary. The samples are encoded as described for the `readsamps` return value.

getfillpoint *()* -- Method on audio port object

Return the `fill point'.

setfillpoint *(fillpoint)* -- Method on audio port object

Set the `fill point'.

getconfig *()* -- Method on audio port object

Return a configuration object containing the current configuration of the port.

setconfig *(config)* -- Method on audio port object

Set the configuration from the argument, a configuration object.

getstatus *(list)* -- Method on audio port object

Get status information on last error.

Prev: Configuration Objects Up: al Top: Top

macspeech -- Python library reference

Next: EasyDialogs Prev: mactcp Up: Macintosh Specific Services Top: Top

14.10. Built-in Module macspeech

This module provides an interface to the Macintosh Speech Manager, allowing you to let the Macintosh utter phrases. You need a version of the speech manager extension (version 1 and 2 have been tested) in your `Extensions` folder for this to work. The module does not provide full access to all features of the Speech Manager yet. It may not be available in all Mac Python versions.

Available *()* -- function of module macspeech

> Test availability of the Speech Manager extension (and, on the PowerPC, the Speech Manager shared library). Return 0 or 1.

Version *()* -- function of module macspeech

> Return the (integer) version number of the Speech Manager.

SpeakString *(str)* -- function of module macspeech

> Utter the string *str* using the default voice, asynchronously. This aborts any speech that may still be active from prior `SpeakString` invocations.

Busy *()* -- function of module macspeech

> Return the number of speech channels busy, system-wide.

CountVoices *()* -- function of module macspeech

> Return the number of different voices available.

GetIndVoice *(num)* -- function of module macspeech

> Return a voice object for voice number *num*.

Menu

Next: EasyDialogs Prev: mactcp Up: Macintosh Specific Services Top: Top

uu -- Python library reference

11.13. Standard module uu

This module encodes and decodes files in uuencode format, allowing arbitrary binary data to be transferred over ascii-only connections. Whereever a file argument is expected, the methods accept either a pathname (`'-'` for stdin/stdout) or a file-like object.

Normally you would pass filenames, but there is one case where you have to open the file yourself: if you are on a non-unix platform and your binary file is actually a textfile that you want encoded unix-compatible you will have to open the file yourself as a textfile, so newline conversion is performed.

This code was contributed by Lance Ellinghouse, and modified by Jack Jansen.

The `uu` module defines the following functions:

encode *(in_file, out_file[, name, mode])* -- function of module uu

> Uuencode file *in_file* into file *out_file*. The uuencoded file will have the header specifying *name* and *mode* as the defaults for the results of decoding the file. The default defaults are taken from *in_file*, or `'-'` and `0666` respectively.

decode *(in_file[, out_file, mode])* -- function of module uu

> This call decodes uuencoded file *in_file* placing the result on file *out_file*. If *out_file* is a pathname the *mode* is also set. Defaults for *out_file* and *mode* are taken from the uuencode header.

Generic Operating System Services -- Python library reference

Next: Optional Operating System Services Prev: Miscellaneous Services Up: Top Top: Top

6. Generic Operating System Services

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

os

--- Miscellaneous OS interfaces.

time

--- Time access and conversions.

getopt

--- Parser for command line options.

tempfile

--- Generate temporary file names.

Menu

Next: Optional Operating System Services Prev: Miscellaneous Services Up: Top Top: Top

Mutable Sequence Types -- Python library reference

2.1.5.2. 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(s):len(s)] = [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]` if `i >= 0`

`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 a list 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.

Prev: More String Operations Up: Sequence Types Top: Top

Next: Reference Manual Prev: Instant Users Manual Up: The Python Profiler Top: Top

10.4. What Is Deterministic Profiling?

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

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

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

Next: Reference Manual Prev: Instant Users Manual Up: The Python Profiler Top: Top

Functions -- Python library reference

2.1.7.3. 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).

macdnr -- Python library reference

14.5. Built-in Module macdnr

This module provides an interface to the Macintosh Domain Name Resolver. It is usually used in conjunction with the *mactcp* module, to map hostnames to IP-addresses. It may not be available in all Mac Python versions.

The `macdnr` module defines the following functions:

Open *([filename])* -- function of module macdnr

> Open the domain name resolver extension. If *filename* is given it should be the pathname of the extension, otherwise a default is used. Normally, this call is not needed since the other calls will open the extension automatically.

Close *()* -- function of module macdnr

> Close the resolver extension. Again, not needed for normal use.

StrToAddr *(hostname)* -- function of module macdnr

> Look up the IP address for *hostname*. This call returns a dnr result object of the ``address'' variation.

AddrToName *(addr)* -- function of module macdnr

> Do a reverse lookup on the 32-bit integer IP-address *addr*. Returns a dnr result object of the ``address'' variation.

AddrToStr *(addr)* -- function of module macdnr

> Convert the 32-bit integer IP-address *addr* to a dotted-decimal string. Returns the string.

HInfo *(hostname)* -- function of module macdnr

> Query the nameservers for a `HInfo` record for host *hostname*. These records contain hardware and software information about the machine in question (if they are available in the first place). Returns a dnr result object of the ``hinfo'' variety.

MXInfo *(domain)* -- function of module macdnr

> Query the nameservers for a mail exchanger for *domain*. This is the hostname of a host willing to accept SMTP mail for the given domain. Returns a dnr result object of the ``mx'' variety.

Menu

Profiler Changes -- Python library reference

10.2. How Is This Profiler Different From The Old Profiler?

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

To be specific:

Bugs removed:

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

Accuracy increased:

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

Speed increased:

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

Recursive functions support:

Cumulative times in recursive functions are correctly calculated; recursive entries are counted.

Large growth in report generating UI:

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

FrameWork -- Python library reference

14.12. Standard module FrameWork

The `FrameWork` module contains classes that together provide a framework for an interactive Macintosh application. The programmer builds an application by creating subclasses that override various methods of the bases classes, thereby implementing the functionality wanted. Overriding functionality can often be done on various different levels, i.e. to handle clicks in a single dialog window in a non-standard way it is not necessary to override the complete event handling.

The `FrameWork` is still very much work-in-progress, and the documentation describes only the most important functionality, and not in the most logical manner at that. Examine the source for more esoteric needs.

The `EasyDialogs` module defines the following functions:

Application *()* -- function of module FrameWork

> An object representing the complete application. See below for a description of the methods. The default `__init__` routine creates an empty window dictionary and a menu bar with an apple menu.

MenuBar *()* -- function of module FrameWork

> An object representing the menubar. This object is usually not created by the user.

Menu *(bar, title[, after])* -- function of module FrameWork

> An object representing a menu. Upon creation you pass the `MenuBar` the menu appears in, the *title* string and a position (1-based) *after* where the menu should appear (default: at the end).

MenuItem *(menu, title[, shortcut, callback])* -- function of module FrameWork

> Create a menu item object. The arguments are the menu to crate the item it, the item title string and optionally the keyboard shortcut and a callback routine. The callback is called with the arguments menu-id, item number within menu (1-based), current front window and the event record.

Separator *(menu)* -- function of module FrameWork

> Add a separator to the end of a menu.

SubMenu *(menu, label)* -- function of module FrameWork

> Create a submenu named *label* under menu *menu*. The menu object is returned.

Window *(parent)* -- function of module FrameWork

> Creates a (modeless) window. *Parent* is the application object to which the window belongs. The window is not displayed until later.

DialogWindow *(parent)* -- function of module FrameWork

> Creates a modeless dialog window.

Menu

Calibration -- Python library reference

Next: Profiler Extensions Prev: Limitations Up: The Python Profiler Top: Top

10.7. Calibration

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

```
import profile

pr = profile.Profile()

pr.calibrate(100)

pr.calibrate(100)

pr.calibrate(100)
```

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

```
pr.calibrate(1000)
```

or even:

```
pr.calibrate(10000)
```

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

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

```
def trace_dispatch(self, frame, event, arg):

    t = self.timer()

    t = t[0] + t[1] - self.t - .00053 # Calibration constant
```

```
if self.dispatch[event](frame,t):

    t = self.timer()

    self.t = t[0] + t[1]

else:

    r = self.timer()

    self.t = r[0] + r[1] - t # put back unrecorded delta

return
```

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

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

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

Next: Profiler Extensions Prev: Limitations Up: The Python Profiler Top: Top

Socket Example -- Python library reference

Prev: Socket Objects Up: socket Top: Top

7.2.2. Example

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

```
# Echo server program

from socket import *

HOST = ''                 # Symbolic name meaning the local host

PORT = 50007              # Arbitrary non-privileged server

s = socket(AF_INET, SOCK_STREAM)

s.bind(HOST, PORT)

s.listen(1)

conn, addr = s.accept()

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`
```

sgmllib -- Python library reference

11.9. Standard Module sgmllib

This module defines a class `SGMLParser` which serves as the basis for parsing text files formatted in SGML (Standard Generalized Mark-up Language). In fact, it does not provide a full SGML parser --- it only parses SGML insofar as it is used by HTML, and the module only exists as a basis for the `htmllib` module. In particular, the parser is hardcoded to recognize the following elements:

Opening and closing tags of the form ``<*tag attr="value" ...*>'' and ``</*tag*>'', respectively.

Character references of the form ``&#*name*;''.

Entity references of the form ``&*name*;''.

SGML comments of the form ``<!--*text*>''.

The `SGMLParser` class must be instantiated without arguments. It has the following interface methods:

reset *()* -- function of module sgmllib

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

setnomoretags *()* -- function of module sgmllib

Stop processing tags. Treat all following input as literal input (CDATA). (This is only provided so the HTML tag `<PLAINTEXT>` can be implemented.)

setliteral *()* -- function of module sgmllib

Enter literal mode (CDATA mode).

feed *(data)* -- function of module sgmllib

Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called.

close *()* -- function of module sgmllib

Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call `SGMLParser.close()`.

handle_charref *(ref)* -- function of module sgmllib

This method is called to process a character reference of the form ``&#*ref*;'' where *ref* is a decimal number in the range 0-255. It translates the character to ASCII and calls the method `handle_data()` with the character as argument. If *ref* is invalid or out of range, the method `unknown_charref(ref)` is called instead.

handle_entityref *(ref)* -- function of module sgmllib

This method is called to process an entity reference of the form ``&*ref*;'' where *ref* is an alphabetic entity reference. It looks for *ref* in the instance (or class) variable `entitydefs` which should give the entity's translation. If a translation is found, it calls the method `handle_data()` with the translation; otherwise, it calls the method `unknown_entityref(ref)`.

handle_data *(data)* -- function of module sgmllib

This method is called to process arbitrary data. It is intended to be overridden by a derived class; the base class implementation does nothing.

unknown_starttag *(tag, attributes)* -- function of module sgmllib

> This method is called to process an unknown start tag. It is intended to be overridden by a derived class; the base class implementation does nothing. The *attributes* argument is a list of (*name*, *value*) pairs containing the attributes found inside the tag's <> brackets. The *name* has been translated to lower case and double quotes and backslashes in the *value* have been interpreted. For instance, for the tag <A HREF="http://www.cwi.nl/">, this method would be called as `unknown_starttag('a', [('href', 'http://www.cwi.nl/')])`.

unknown_endtag *(tag)* -- function of module sgmllib

> This method is called to process an unknown end tag. It is intended to be overridden by a derived class; the base class implementation does nothing.

unknown_charref *(ref)* -- function of module sgmllib

> This method is called to process an unknown character reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

unknown_entityref *(ref)* -- function of module sgmllib

> This method is called to process an unknown entity reference. It is intended to be overridden by a derived class; the base class implementation does nothing.

Apart from overriding or extending the methods listed above, derived classes may also define methods of the following form to define processing of specific tags. Tag names in the input stream are case independent; the *tag* occurring in method names must be in lower case:

start_*tag* *(attributes)* -- function of module sgmllib

> This method is called to process an opening tag *tag*. It has preference over `do_tag()`. The *attributes* argument has the same meaning as described for `unknown_tag()` above.

do_*tag* *(attributes)* -- function of module sgmllib

> This method is called to process an opening tag *tag* that does not come with a matching closing tag. The *attributes* argument has the same meaning as described for `unknown_tag()` above.

end_*tag* *()* -- function of module sgmllib

> This method is called to process a closing tag *tag*.

Note that the parser maintains a stack of opening tags for which no matching closing tag has been found yet. Only tags processed by `start_tag()` are pushed on this stack. Definition of a `end_tag()` method is optional for these tags. For tags processed by `do_tag()` or by `unknown_tag()`, no `end_tag()` method must be defined.

Next: rfc822 Prev: htmllib Up: Internet and WWW Top: Top

jpeg -- Python library reference

12.4. Built-in Module jpeg

The module `jpeg` provides access to the jpeg compressor and decompressor written by the Independent JPEG Group. JPEG is a (draft?) standard for compressing pictures. For details on jpeg or the Independent JPEG Group software refer to the JPEG standard or the documentation provided with the software.

The `jpeg` module defines these functions:

compress *(data, w, h, b)* -- function of module jpeg

> Treat data as a pixmap of width *w* and height *h*, with *b* bytes per pixel. The data is in SGI GL order, so the first pixel is in the lower-left corner. This means that `lrectread` return data can immediately be passed to compress. Currently only 1 byte and 4 byte pixels are allowed, the former being treated as greyscale and the latter as RGB color. Compress returns a string that contains the compressed picture, in JFIF format.

decompress *(data)* -- function of module jpeg

> 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)* -- function of module jpeg

> 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.

The Null Object -- Python library reference

2.1.7.7. 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`.

Variable Index -- Python library reference

Next: Module Index Prev: Function Index Up: Top Top: Top

Variable Index

Next: Module Index Prev: Function Index Up: Top Top: Top

grp -- Python library reference

8.4. Built-in Module grp

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

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

It defines the following items:

getgrgid *(gid)* -- function of module grp

> Return the group database entry for the given numeric group ID.

getgrnam *(name)* -- function of module grp

> Return the group database entry for the given group name.

getgrall *()* -- function of module grp

> Return a list of all available group entries, in arbitrary order.

whrandom -- Python library reference

Next: array Prev: rand Up: Miscellaneous Services Top: Top

5.3. Standard Module whrandom

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

random *()* -- function of module whrandom

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

seed *(x, y, z)* -- function of module whrandom

> 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.

Next: array Prev: rand Up: Miscellaneous Services Top: Top

pickle -- Python library reference

3.4. Standard Module pickle

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

recursive objects (objects containing references to themselves)

object sharing (references to the same object in different places)

user-defined classes and their instances

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

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

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

There are some restrictions on the pickling of class instances.

First of all, the class must be defined at the top level in a module.

Next, it must normally be possible to create class instances by calling the class without arguments. If this is undesirable, the class can define a method __getinitargs__(), which should return a *tuple* containing the arguments to be passed to the class constructor (__init__()). Classes can further influence how their instances are pickled --- if the class defines the method __getstate__(), it is called and the return state is pickled as the contents for the instance, and if the class defines the method __setstate__(), it is called with the unpickled state. (Note that these methods can also be used to implement copying class instances.) If there is no `__getstate__()` method, the instance's `__dict__` is pickled. If there is no `__setstate__()` method, the pickled object must be a dictionary and its items are assigned to the new instance's dictionary. (If a class defines both `__getstate__()` and `__setstate__()`, the state object needn't be a dictionary --- these methods can do what they want.) This protocol is also used by the shallow and deep copying operations defined in the `copy` module. Note that when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

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

The interface can be summarized as follows.

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

```
p = pickle.Pickler(f)
```

p.dump(x)

A shorthand for this is:

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

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

```
u = pickle.Unpickler(f)
```

x = u.load()

A shorthand is:

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

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

`None`

integers, long integers, floating point numbers

strings

tuples, lists and dictionaries containing only picklable objects classes that are defined at the top level in a module

instances of such classes whose `__dict__` or `__setstate__()` is picklable

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

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

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

dump *(object, file)* -- function of module pickle

> Write a pickled representation of *obect* to the open file object *file*. This is equivalent to `Pickler(file).dump(object)`.

load *(file)* -- function of module pickle

> Read a pickled object from the open file object *file*. This is equivalent to `Unpickler(file).load()`.

dumps *(object)* -- function of module pickle

> Return the pickled representation of the object as a string, instead of writing it to a file.

loads *(string)* -- function of module pickle

> Read a pickled object from a string instead of a file. Characters in the string past the pickled object's representation are ignored.

PicklingError -- exception of module pickle

> This exception is raised when an unpicklable object is passed to `Pickler.dump()`.

mimetools.Message Methods -- Python library reference

11.11.1. Additional Methods of Message objects

The `mimetools.Message` class defines the following methods in addition to the `rfc822.Message` class:

getplist *()* -- Method on mimetool.Message

> Return the parameter list of the `Content-type` header. This is a list if strings. For parameters of the form `*key=value*', *key* is converted to lower case but *value* is not. For example, if the message contains the header ``Content-type: text/html; spam=1; Spam=2; Spam`' then `getplist()` will return the Python list `['spam=1', 'spam=2', 'Spam']`.

getparam *(name)* -- Method on mimetool.Message

> Return the *value* of the first parameter (as returned by `getplist()` of the form `*name=value*' for the given *name*. If *value* is surrounded by quotes of the form <...> or "...", these are removed.

getencoding *()* -- Method on mimetool.Message

> Return the encoding specified in the ``Content-transfer-encoding`' message header. If no such header exists, return `"7bit"`. The encoding is converted to lower case.

gettype *()* -- Method on mimetool.Message

> Return the message type (of the form `*type/varsubtype*') as specified in the ``Content-type`' header. If no such header exists, return `"text/plain"`. The type is converted to lower case.

getmaintype *()* -- Method on mimetool.Message

> Return the main type as specified in the ``Content-type`' header. If no such header exists, return `"text"`. The main type is converted to lower case.

getsubtype *()* -- Method on mimetool.Message

> Return the subtype as specified in the ``Content-type`' header. If no such header exists, return `"plain"`. The subtype is converted to lower case.

Next: fm Prev: FL (uppercase) Up: SGI IRIX Specific Services Top: Top

16.6. Standard Module flp

This module defines functions that can read form definitions created by the `form designer' (`fdesign`) program that comes with the FORMS library (see module `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!

Next: fm Prev: FL (uppercase) Up: SGI IRIX Specific Services Top: Top

alias objects -- Python library reference

14.6.2. alias objects

data -- attribute of alias object

> The raw data for the Alias record, suitable for storing in a resource or transmitting to other programs.

Resolve *([file])* -- Method on alias object

> Resolve the alias. If the alias was created as a relative alias you should pass the file relative to which it is. Return the FSSpec for the file pointed to and a flag indicating whether the alias object itself was modified during the search process.

GetInfo *(num)* -- Method on alias object

> An interface to the C routine `GetAliasInfo()`.

Update *(file, [file2])* -- Method on alias object

> Update the alias to point to the *file* given. If *file2* is present a relative alias will be created.

Note that it is currently not possible to directly manipulate a resource as an alias object. Hence, after calling *Update* or after *Resolve* indicates that the alias has changed the Python program is responsible for getting the *data* from the alias object and modifying the resource.

Cryptographic Services -- Python library reference

13. Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here's an overview:

md5

--- RSA's MD5 message digest algorithm.

mpz

--- Interface to the GNU MP library for arbitrary precision arithmetic.

rotor

--- Enigma-like encryption and decryption.

Hardcore cypherpunks will probably find the Python Cryptography Kit of further interest; the package adds built-in modules for DES and IDEA encryption, and provides a Python module for reading and decrypting PGP files. The Python Cryptography Kit is not distributed with Python but available separately. See the URL http://www.cs.mcgill.ca/%7Efnord/crypt.html for more information.

Menu

UNIX Specific Services -- Python library reference

Next: The Python Debugger Prev: Optional Operating System Services Up: Top Top: Top

8. UNIX Specific Services

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

posix

--- The most common Posix system calls (normally used via module `os`).

posixpath

--- Common Posix pathname manipulations (normally used via `os.path`).

pwd

--- The password database (`getpwnam()` and friends).

grp

--- The group database (`getgrnam()` and friends).

dbm

--- The standard ``database'' interface, based on `ndbm`.

gdbm

--- GNU's reinterpretation of dbm.

termios

--- Posix style tty control.

fcntl

--- The `fcntl()` and `ioctl()` system calls.

posixfile

--- A file-like object with support for locking.

Menu

posix -- Python library reference

8.1. Built-in Module posix

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

Do not import this module directly. Instead, import the module `os`, which provides a *portable* version of this interface. On UNIX, the `os` module provides a superset of the `posix` interface. On non-UNIX operating systems the `posix` module is not available, but a subset is always available through the `os` interface. Once `os` is imported, there is *no* performance penalty in using it instead of `posix`. The descriptions below are very terse; refer to the corresponding UNIX manual entry for more information. Arguments called *path* refer to a pathname given as a string.

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

Module `posix` defines the following data items:

environ -- data of module posix

> A dictionary representing the string environment at the time the interpreter was started. For example, `posix.environ['HOME']` is the pathname of your home directory, equivalent to `getenv("HOME")` in C. Modifying this dictionary does not affect the string environment passed on by `execv()`, `popen()` or `system()`; if you need to change the environment, pass `environ` to `execve()` or add variable assignments and export statements to the command string for `system()` or `popen()`.(1)

error -- exception of module posix

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

It defines the following functions and constants:

chdir *(path)* -- function of module posix

> Change the current working directory to *path*.

chmod *(path, mode)* -- function of module posix

> Change the mode of *path* to the numeric *mode*.

chown *(path, uid, gid)* -- function of module posix

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

close *(fd)* -- function of module posix

> Close file descriptor *fd*.

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

dup *(fd)* -- function of module posix

> Return a duplicate of file descriptor *fd*.

dup2 *(fd, fd2)* -- function of module posix

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. Return `None`.

execv *(path, args)* -- function of module posix

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)* -- function of module posix

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)* -- function of module posix

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[, bufsize]])* -- function of module posix

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

fork *()* -- function of module posix

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

fstat *(fd)* -- function of module posix

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

getcwd *()* -- function of module posix

Return a string representing the current working directory.

getegid *()* -- function of module posix

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

geteuid *()* -- function of module posix

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

getgid *()* -- function of module posix

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

getpid *()* -- function of module posix

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

getppid *()* -- function of module posix

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

getuid *()* -- function of module posix

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

kill *(pid, sig)* -- function of module posix

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

link *(src, dst)* -- function of module posix

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

listdir *(path)* -- function of module posix

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)* -- function of module posix

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)* -- function of module posix

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

mkdir *(path, mode)* -- function of module posix

Create a directory named *path* with numeric mode *mode*.

nice *(increment)* -- function of module posix

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

open *(file, flags, mode)* -- function of module posix

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

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

pipe *()* -- function of module posix

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

popen *(command[, mode[, bufsize]])* -- function of module posix

Open a pipe to or from *command*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *bufsize* argument has the same meaning as the corresponding argument to the built-in `open()` function. (Not on MS-DOS.)

read *(fd, n)* -- function of module posix

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

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

readlink *(path)* -- function of module posix

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)* -- function of module posix

Rename the file or directory *src* to *dst*.

rmdir *(path)* -- function of module posix

Remove the directory *path*.

setgid *(gid)* -- function of module posix

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

setuid *(uid)* -- function of module posix

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

stat *(path)* -- function of module posix

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)* -- function of module posix

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

system *(command)* -- function of module posix

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 *()* -- function of module posix

Return a 5-tuple of floating point numbers indicating accumulated (CPU or other) times, in seconds. The items are: user time, system time, children's user time, children's system time, and elapsed real time since a fixed point in the past, in that order. See the UNIX manual page *times*(2). (Not on MS-DOS.)

umask *(mask)* -- function of module posix

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

uname *()* -- function of module posix

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

unlink *(path)* -- function of module posix

Unlink *path*.

utime *(path, (atime, mtime))* -- function of module posix

Set the access and modified time of the file to the given values. (The second argument is a tuple of two items.)

wait *()* -- function of module posix

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)* -- function of module posix

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)* -- function of module posix

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

Note: this function is intended for low-level I/O and must be applied to a file descriptor as returned by `posix.open()` or `posix.pipe()`. To write a ``file object'' returned by the built-in function `open` or by `posix.popen` or `posix.fdopen`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

WNOHANG -- data of module posix

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

---------- Footnotes ----------

(1) The problem with automatically passing on `environ` is that there is no portable way of changing the environment.

termios -- Python library reference

8.7. Built-in Module termios

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

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

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

The module defines the following functions:

tcgetattr *(fd)* -- function of module termios

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

tcsetattr *(fd, when, attributes)* -- function of module termios

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

tcsendbreak *(fd, duration)* -- function of module termios

> Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25--0.5 seconds; a nonzero *duration* has a system dependent meaning.

tcdrain *(fd)* -- function of module termios

> Wait until all output written to file descriptor *fd* has been transmitted.

tcflush *(fd, queue)* -- function of module termios

> Discard queued data on file descriptor *fd*. The *queue* selector specifies which queue: `TERMIOS.TCIFLUSH` for the input queue, `TERMIOS.TCOFLUSH` for the output queue, or `TERMIOS.TCIOFLUSH` for both queues.

tcflow *(fd, action)* -- function of module termios

> Suspend or resume input or output on file descriptor *fd*. The *action* argument can be `TERMIOS.TCOOFF` to suspend output, `TERMIOS.TCOON` to restart output, `TERMIOS.TCIOFF` to suspend input, or `TERMIOS.TCION` to restart input.

Menu

SunOS Specific Services -- Python library reference

Next: Function Index Prev: SGI IRIX Specific Services Up: Top Top: Top

17. SunOS Specific Services

The modules described in this chapter provide interfaces to features that are unique to the SunOS operating system (versions 4 and 5; the latter is also known as Solaris version 2).

Menu

STDWIN Functions -- Python library reference

15.1.1. Functions Defined in Module stdwin

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

open *(title)* -- function of module stdwin

> 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 *()* -- function of module stdwin

> 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 *()* -- function of module stdwin

> Return the next event, if one is immediately available. If no event is available, return `()`.

getactive *()* -- function of module stdwin

> 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)* -- function of module stdwin

> Return the list of font names in the system that match the pattern (a string). The pattern should normally be `'*'`; returns all available fonts. If the underlying window system is X11, other patterns follow the standard X11 font selection syntax (as used e.g. in resource definitions), i.e. the wildcard character `'*'` matches any sequence of characters (including none) and `'?'` matches any single character. On the Macintosh this function currently returns an empty list.

setdefscrollbars *(hflag, vflag)* -- function of module stdwin

> Set the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll bars.

setdefwinpos *(h, v)* -- function of module stdwin

> Set the default window position for windows opened subsequently.

setdefwinsize *(width, height)* -- function of module stdwin

> Set the default window size for windows opened subsequently.

getdefscrollbars *()* -- function of module stdwin

> Return the flags controlling whether subsequently opened windows will have horizontal and/or vertical scroll bars.

getdefwinpos *()* -- function of module stdwin

> Return the default window position for windows opened subsequently.

getdefwinsize *()* -- function of module stdwin

> Return the default window size for windows opened subsequently.

getscrsize *()* -- function of module stdwin

> Return the screen size in pixels.

getscrmm *()* -- function of module stdwin

Return the screen size in millimeters.

fetchcolor *(colorname)* -- function of module stdwin

Return the pixel value corresponding to the given color name. Return the default foreground color for unknown color names. Hint: the following code tests whether you are on a machine that supports more than two colors:

```
if stdwin.fetchcolor('black') <> \
      stdwin.fetchcolor('red') <> \
      stdwin.fetchcolor('white'):
   print 'color machine'
else:
   print 'monochrome machine'
```

setfgcolor *(pixel)* -- function of module stdwin

Set the default foreground color. This will become the default foreground color of windows opened subsequently, including dialogs.

setbgcolor *(pixel)* -- function of module stdwin

Set the default background color. This will become the default background color of windows opened subsequently, including dialogs.

getfgcolor *()* -- function of module stdwin

Return the pixel value of the current default foreground color.

getbgcolor *()* -- function of module stdwin

Return the pixel value of the current default background color.

setfont *(fontname)* -- function of module stdwin

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)* -- function of module stdwin

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)* -- function of module stdwin

Create a new bitmap object of the given dimensions. Methods of bitmap objects are described below. Not available on the Macintosh.

fleep *()* -- function of module stdwin

Cause a beep or bell (or perhaps a `visual bell' or flash, hence the name).

message *(string)* -- function of module stdwin

Display a dialog box containing the string. The user must click OK before the function returns.

askync *(prompt, default)* -- function of module stdwin

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)* -- function of module stdwin

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)* -- function of module stdwin

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)* -- function of module stdwin

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)* -- function of module stdwin

Return the contents of the system's cut buffer number *i*.

rotatecutbuffers *(n)* -- function of module stdwin

On X11, rotate the 8 cut buffers by *n*. Ignored on the Macintosh.

getselection *(i)* -- function of module stdwin

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)* -- function of module stdwin

Reset selection number *i*, if this process owns it. (See window method `setselection()`).

baseline *()* -- function of module stdwin

Return the baseline of the current font (defined by STDWIN as the vertical distance between the baseline and the top of the characters).

lineheight *()* -- function of module stdwin

Return the total line height of the current font.

textbreak *(str, width)* -- function of module stdwin

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)* -- function of module stdwin

Return the width in bits of the string when drawn in the current font.

connectionnumber *()* -- function of module stdwin

fileno *()* -- function of module stdwin

(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.

---------- Footnotes ----------

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

Next: Window Objects Prev: stdwin Up: stdwin Top: Top

Code Objects -- Python library reference

2.1.7.5. Code Objects

Code objects are used by the implementation to represent ``pseudo-compiled'' executable Python code such as a function body. They differ from function objects because they don't contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `func_code` attribute. A code object can be executed or evaluated by passing it (instead of a source string) to the `exec` statement or the built-in `eval()` function. (See the Python Reference Manual for more info.)

Type Objects -- Python library reference

2.1.7.6. 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. The standard module `types` defines names for all standard built-in types. Types are written like this: `<type 'int'>`.

Numeric Types -- Python library reference

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.(1) The functions `int()`, `long()` and `float()` can be used to coerce numbers to a specific type. All numeric types support the following operations, sorted by ascending priority (operations in the same box have the same priority; all numeric operations have a higher priority than comparison operations):

Operation

Result --- *Notes*

`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* --- (1)

`x % y`

remainder of `x / y``-x`

x negated

`+x`

x unchanged`abs(x)`

absolute value of *x*

`int(x)`

x converted to integer --- (2)

`long(x)`

x converted to long integer --- (2)

`float(x)`

x converted to floating point

`divmod(x, y)`

the pair `(x / y, x % y)` --- (3)

`pow(`*`x, y`*`)`

x to the power *y*

Notes:

(1)

For (plain or long) integer division, the result is an integer. The result is always rounded towards minus infinity: 1/2 is 0, (-1)/2 is -1, 1/(-2) is -1, and (-1)/(-2) is 0.

(2)

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

(3)

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

Menu

Bit-string Operations

---------- Footnotes ----------

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

traceback -- Python library reference

3.3. Standard Module traceback

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

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

The module defines the following functions:

print_tb *(traceback[, limit])* -- function of module traceback

> Print up to *limit* stack trace entries from *traceback*. If *limit* is omitted or `None`, all entries are printed.

extract_tb *(traceback[, limit])* -- function of module traceback

> Return a list of up to *limit* ``pre-processed'' stack trace entries extracted from *traceback*. It is useful for alternate formatting of stack traces. If *limit* is omitted or `None`, all entries are extracted. A ``pre-processed'' stack trace entry is a quadruple (*filename*, *line number*, *function name*, *line text*) representing the information that is usually printed for a stack trace. The *line text* is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

print_exception *(type, value, traceback[, limit])* -- function of module traceback

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

print_exc *([limit])* -- function of module traceback

> This is a shorthand for `print_exception(sys.exc_type, sys.exc_value, sys.exc_traceback, limit)`.

print_last *([limit])* -- function of module traceback

> This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit)`.

Socket Objects -- Python library reference

Next: Socket Example Prev: socket Up: socket Top: Top

7.2.1. Socket Objects

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

accept *()* -- Method on socket

> 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)* -- Method on socket

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

close *()* -- Method on socket

> 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)* -- Method on socket

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

fileno *()* -- Method on socket

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

getpeername *()* -- Method on socket

> 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. (The format of the address returned depends on the address family --- see above.) On some systems this function is not supported.

getsockname *()* -- Method on socket

> Return the socket's own address. This is useful to find out the port number of an IP socket, for instance. (The format of the address returned depends on the address family --- see above.)

getsockopt *(level, optname[, buflen])* -- Method on socket

> Return the value of the given socket option (see the UNIX man page *getsockopt*(2)). The needed symbolic constants (`SO_*` etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a string. It is up to the caller to decode the contents of the buffer (see the optional built-in module `struct` for a way to decode C structures encoded as strings).

listen *(backlog)* -- Method on socket

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

makefile *([mode[, bufsize]])* -- Method on socket

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. The optional *mode* and *bufsize* arguments are interpreted the same way as by the built-in `open()` function.

recv *(bufsize[, flags])* -- Method on socket

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

recvfrom *(bufsize[, flags])* -- Method on socket

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

send *(string[, flags])* -- Method on socket

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

sendto *(string[, flags], address)* -- Method on socket

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

setblocking *(flag)* -- Method on socket

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

setsockopt *(level, optname, value)* -- Method on socket

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

shutdown *(how)* -- Method on socket

> 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.

Next: Socket Example Prev: socket Up: socket Top: Top

ftplib -- Python library reference

11.4. Standard Module ftplib

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

Here's a sample session using the `ftplib` module:

```
>>> from ftplib import FTP

>>> ftp = FTP('ftp.cwi.nl')    # connect to host, default port

>>> ftp.login()                # user anonymous, passwd user@hostname

>>> ftp.retrlines('LIST')      # list directory contents

total 24418

drwxrwsr-x   5 ftp-usr  pdmaint     1536 Mar 20 09:48 .

dr-xr-srwt 105 ftp-usr  pdmaint     1536 Mar 21 14:32 ..

-rw-r--r--   1 ftp-usr  pdmaint     5305 Mar 20 09:48 INDEX

 .

 .

 .

>>> ftp.quit()
```

The module defines the following items:

FTP *([host[, user, passwd, acct]])* -- function of module ftplib

> Return a new instance of the `FTP` class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made

(where *passwd* and *acct* default to the empty string when not given).

all_errors -- data of module ftplib

The set of all exceptions (as a tuple) that methods of `FTP` instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed below as well as `socket.error` and `IOError`.

error_reply -- exception of module ftplib

Exception raised when an unexpected reply is received from the server.

error_temp -- exception of module ftplib

Exception raised when an error code in the range 400--499 is received.

error_perm -- exception of module ftplib

Exception raised when an error code in the range 500--599 is received.

error_proto -- exception of module ftplib

Exception raised when a reply is received from the server that does not begin with a digit in the range 1--5.

Menu

FTP Objects

Top -- Python library reference

The Python library

This file 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. (These manuals are not yet available in INFO or Texinfo format.)

This version corresponds to Python version 1.3 (13 Oct 1995).

Menu

os -- Python library reference

6.1. Standard Module os

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

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

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

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

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

name -- data of module os

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

path -- data of module os

> 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 -- data of module os

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

pardir -- data of module os

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

sep -- data of module os

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

pathsep -- data of module os

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

defpath -- data of module os

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

execl *(path, arg0, arg1, ...)* -- function of module os

> This is equivalent to `os.execv(`*path*`, (`*arg0*`, `*arg1*`, ...))`.

execle *(path, arg0, arg1, ..., env)* -- function of module os

> This is equivalent to `os.execve(`*`path, (arg0, arg1, ...), env)`*.

execlp *(path, arg0, arg1, ...)* -- function of module os

> This is equivalent to `os.execvp(`*`path, (arg0, arg1, ...))`*.

execvp *(path, args)* -- function of module os

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

execvpe *(path, args, env)* -- function of module os

> This is a cross between `os.execve()` and `os.execvp()`. The directory list is obtained from *`env['PATH']`*.

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

console window object -- Python library reference

14.4.2. console window object

file -- attribute of console window

> The file object corresponding to this console window. If the file is buffered, you should call `file.flush()` between `write()` and `read()` calls.

setmode *(mode)* -- Method on console window

> Set the input mode of the console to *C_ECHO*, etc.

settabs *(n)* -- Method on console window

> Set the tabsize to *n* spaces.

cleos *()* -- Method on console window

> Clear to end-of-screen.

cleol *()* -- Method on console window

> Clear to end-of-line.

inverse *(onoff)* -- Method on console window

> Enable inverse-video mode: characters with the high bit set are displayed in inverse video (this disables the upper half of a non-ASCII character set).

gotoxy *(x, y)* -- Method on console window

> Set the cursor to position `(x, y)`.

hide *()* -- Method on console window

> Hide the window, remembering the contents.

show *()* -- Method on console window

> Show the window again.

echo2printer *()* -- Method on console window

> Copy everything written to the window to the printer as well.

sys -- Python library reference

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 -- data of module sys

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

builtin_module_names -- data of module sys

> 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 -- data of module sys

exc_value -- data of module sys

exc_traceback -- data of module sys

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

exit *(n)* -- function of module sys

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

exitfunc -- data of module sys

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

last_type -- data of module sys

last_value -- data of module sys

last_traceback -- data of module sys

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

modules -- data of module sys

> Gives the list of modules that have already been loaded. This can be manipulated to force

reloading of modules and other tricks.

path -- data of module sys

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

platform -- data of module sys

This string contains a platform identifier. This can be used to append platform-specific components to `sys.path`, for instance.

ps1 -- data of module sys

ps2 -- data of module sys

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`.

setcheckinterval *(interval)* -- function of module sys

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

settrace *(tracefunc)* -- function of module sys

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

setprofile *(profilefunc)* -- function of module sys

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

stdin -- data of module sys

stdout -- data of module sys

stderr -- data of module sys

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. (Changing these objects doesn't affect the standard I/O streams of processes executed by `popen()`, `system()` or the `exec*()` family of functions in the `os` module.)

tracebacklimit -- data of module sys

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.

Next: types Prev: Python Services Up: Python Services Top: Top

Miscellaneous Services -- Python library reference

Next: Generic Operating System Services Prev: String Services Up: Top Top: Top

5. Miscellaneous Services

The modules described in this chapter provide miscellaneous services that are available in all Python versions. Here's an overview:

math

--- Mathematical functions (`sin()` etc.).

rand

--- Integer random number generator.

whrandom

--- Floating point random number generator.

array

--- Efficient arrays of uniformly typed numeric values.

Menu

Next: Generic Operating System Services Prev: String Services Up: Top Top: Top

15.2. Standard Module stdwinevents

This module defines constants used by STDWIN for event types (`WE_ACTIVATE` etc.), command codes (`WC_LEFT` etc.) and selection types (`WS_PRIMARY` etc.). Read the file for details. Suggested usage is

```
>>> from stdwinevents import *


>>>
```

binascii -- Python library reference

11.14. Built-in Module binascii

The binascii module contains a number of methods to convert between binary and various ascii-encoded binary representations. Normally, you will not use these modules directly but use wrapper modules like *uu* or *hexbin* in stead, this module solely exists because bit-manipuation of large amounts of data is slow in python.

The `binascii` module defines the following functions:

a2b_uu *(string)* -- function of module binascii

> Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

b2a_uu *(data)* -- function of module binascii

> Convert binary data to a line of ascii characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45.

a2b_base64 *(string)* -- function of module binascii

> Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

b2a_base64 *(data)* -- function of module binascii

> Convert binary data to a line of ascii characters in base64 coding. The return value is the converted line, including a newline char. The length of *data* should be at most 57 to adhere to the base64 standard.

a2b_hqx *(string)* -- function of module binascii

> Convert binhex4 formatted ascii data to binary, without doing rle-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

rledecode_hqx *(data)* -- function of module binascii

> Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses `0x90` after a byte as a repeat indicator, followed by a count. A count of `0` specifies a byte value of `0x90`. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the *Incomplete* exception is raised.

rlecode_hqx *(data)* -- function of module binascii

> Perform binhex4 style RLE-compression on *data* and return the result.

b2a_hqx *(data)* -- function of module binascii

> Perform hexbin4 binary-to-ascii translation and return the resulting string. The argument should already be rle-coded, and have a length divisible by 3 (except possibly the last fragment).

crc_hqx *(data, crc)* -- function of module binascii

> Compute the binhex4 crc value of *data*, starting with an initial *crc* and returning the result.

Error -- exception of module binascii

> Exception raised on errors. These are usually programming errors.

Incomplete -- exception of module binascii

Exception raised on incomplete data. These are usually not programming errors, but handled by reading a little more data and trying again.

Prev: uu Up: Internet and WWW Top: Top

termios Example -- Python library reference

8.7.1. Example

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

```
def getpass(prompt = "Password: "):

    import termios, TERMIOS, sys

    fd = sys.stdin.fileno()

    old = termios.tcgetattr(fd)

    new = termios.tcgetattr(fd)

    new[3] = new[3] & ~TERMIOS.ECHO          # lflags

    try:

        termios.tcsetattr(fd, TERMIOS.TCSADRAIN, new)

        passwd = raw_input(prompt)

    finally:

        termios.tcsetattr(fd, TERMIOS.TCSADRAIN, old)

    return passwd
```

voice objects -- Python library reference

14.10.1. voice objects

Voice objects contain the description of a voice. It is currently not yet possible to access the parameters of a voice.

GetGender *()* -- Method on voice object

> Return the gender of the voice: 0 for male, 1 for female and for neuter.

NewChannel *()* -- Method on voice object

> Return a new speech channel object using this voice.

Bitmap Objects -- Python library reference

Next: Text-edit Objects Prev: Menu Objects Up: stdwin Top: Top

15.1.5. Bitmap Objects

A bitmap represents a rectangular array of bits. The top left bit has coordinate (0, 0). A bitmap can be drawn with the `bitmap` method of a drawing object. Bitmaps are currently not available on the Macintosh.

The following methods are defined:

getsize *()* -- Method on bitmap

> 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)* -- Method on bitmap

> Set the value of the bit indicated by *point* to *bit*.

getbit *(point)* -- Method on bitmap

> Return the value of the bit indicated by *point*.

close *()* -- Method on bitmap

> Discard the bitmap object. It should not be used again.

Next: Text-edit Objects Prev: Menu Objects Up: stdwin Top: Top

rfc822 -- Python library reference

11.10. Standard Module rfc822

This module defines a class, `Message`, which represents a collection of ``email headers'' as defined by the Internet standard RFC 822. It is used in various contexts, usually to read such headers from a file.

A `Message` instance is instantiated with an open file object as parameter. Instantiation reads headers from the file up to a blank line and stores them in the instance; after instantiation, the file is positioned directly after the blank line that terminates the headers.

Input lines as read from the file may either be terminated by CR-LF or by a single linefeed; a terminating CR-LF is replaced by a single linefeed before the line is stored.

All header matching is done independent of upper or lower case; e.g. `m['From']`, `m['from']` and `m['FROM']` all yield the same result.

Menu

Exceptions and Error Handling -- Python library reference

3.9.1. Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

ParserError -- exception of module parser

> Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built in `SyntaxError` thrown during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a tuple causing the failure from a parse tree passed to `tuple2ast()` and an explanatory string. Calls to `tuple2ast()` need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions `compileast()`, `expr()`, and `suite()` may throw exceptions which are normally thrown by the parsing and compilation process. These include the built in exceptions `MemoryError`, `OverflowError`, `SyntaxError`, and `SystemError`. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

cd -- Python library reference

16.3. Built-in Module cd

This module provides an interface to the Silicon Graphics CD library. It is available only on Silicon Graphics systems.

The way the library works is as follows. A program opens the CD-ROM device with `cd.open()` and creates a parser to parse the data from the CD with `cd.createparser()`. The object returned by `cd.open()` can be used to read data from the CD, but also to get status information for the CD-ROM device, and to get information about the CD, such as the table of contents. Data from the CD is passed to the parser, which parses the frames, and calls any callback functions that have previously been added.

An audio CD is divided into tracks or programs (the terms are used interchangeably). Tracks can be subdivided into indices. An audio CD contains a table of contents which gives the starts of the tracks on the CD. Index 0 is usually the pause before the start of a track. The start of the track as given by the table of contents is normally the start of index 1.

Positions on a CD can be represented in two ways. Either a frame number or a tuple of three values, minutes, seconds and frames. Most functions use the latter representation. Positions can be both relative to the beginning of the CD, and to the beginning of the track.

Module `cd` defines the following functions and constants:

createparser *()* -- function of module cd

> Create and return an opaque parser object. The methods of the parser object are described below.

msftoframe *(min, sec, frame)* -- function of module cd

> Converts a `(minutes, seconds, frames)` triple representing time in absolute time code into the corresponding CD frame number.

open *([device[, mode]])* -- function of module cd

> Open the CD-ROM device. The return value is an opaque player object; methods of the player object are described below. The device is the name of the SCSI device file, e.g. /dev/scsi/sc0d4l0, or `None`. If omited or `None`, the hardware inventory is consulted to locate a CD-ROM drive. The `mode`, if not omited, should be the string 'r'.

The module defines the following variables:

error -- data of module cd

> Exception raised on various errors.

DATASIZE -- data of module cd

> The size of one frame's worth of audio data. This is the size of the audio data as passed to the callback of type `audio`.

BLOCKSIZE -- data of module cd

> The size of one uninterpreted frame of audio data.

The following variables are states as returned by `getstatus`:

READY -- data of module cd

> The drive is ready for operation loaded with an audio CD.

NODISC -- data of module cd

The drive does not have a CD loaded.

CDROM -- data of module cd

The drive is loaded with a CD-ROM. Subsequent play or read operations will return I/O errors.

ERROR -- data of module cd

An error aoocurred while trying to read the disc or its table of contents.

PLAYING -- data of module cd

The drive is in CD player mode playing an audio CD through its audio jacks.

PAUSED -- data of module cd

The drive is in CD layer mode with play paused.

STILL -- data of module cd

The equivalent of `PAUSED` on older (non 3301) model Toshiba CD-ROM drives. Such drives have never been shipped by SGI.

audio -- data of module cd

pnum -- data of module cd

index -- data of module cd

ptime -- data of module cd

atime -- data of module cd

catalog -- data of module cd

ident -- data of module cd

control -- data of module cd

Integer constants describing the various types of parser callbacks that can be set by the `addcallback()` method of CD parser objects (see below).

Player objects (returned by `cd.open()`) have the following methods:

allowremoval *()* -- Method on CD player object

Unlocks the eject button on the CD-ROM drive permitting the user to eject the caddy if desired.

bestreadsize *()* -- Method on CD player object

Returns the best value to use for the `num_frames` parameter of the `readda` method. Best is defined as the value that permits a continuous flow of data from the CD-ROM drive.

close *()* -- Method on CD player object

Frees the resources associated with the player object. After calling `close`, the methods of the object should no longer be used.

eject *()* -- Method on CD player object

Ejects the caddy from the CD-ROM drive.

getstatus *()* -- Method on CD player object

Returns information pertaining to the current state of the CD-ROM drive. The returned information is a tuple with the following values: `state`, `track`, `rtime`, `atime`, `ttime`, `first`, `last`, `scsi_audio`, `cur_block`. `rtime` is the time relative to the start of the current track; `atime` is the time relative to the beginning of the disc; `ttime` is the total time on the disc. For more information on the meaning of the values, see the manual for CDgetstatus. The value of `state` is one of the following: `cd.ERROR`, `cd.NODISC`, `cd.READY`, `cd.PLAYING`, `cd.PAUSED`, `cd.STILL`, or `cd.CDROM`.

gettrackinfo *(track)* -- Method on CD player object

Returns information about the specified track. The returned information is a tuple consisting of two elements, the start time of the track and the duration of the track.

msftoblock *(min, sec, frame)* -- Method on CD player object

Converts a minutes, seconds, frames triple representing a time in absolute time code into the corresponding logical block number for the given CD-ROM drive. You should use `cd.msftoframe()` rather than `msftoblock()` for comparing times. The logical block number differs from the frame number by an offset required by certain CD-ROM drives.

play *(start, play)* -- Method on CD player object

Starts playback of an audio CD in the CD-ROM drive at the specified track. The audio output appears on the CD-ROM drive's headphone and audio jacks (if fitted). Play stops at the end of the disc. `start` is the number of the track at which to start playing the CD; if `play` is 0, the CD will be set to an initial paused state. The method `togglepause()` can then be used to commence play.

playabs *(min, sec, frame, play)* -- Method on CD player object

Like `play()`, except that the start is given in minutes, seconds, frames instead of a track number.

playtrack *(start, play)* -- Method on CD player object

Like `play()`, except that playing stops at the end of the track.

playtrackabs *(track, min, sec, frame, play)* -- Method on CD player object

Like `play()`, except that playing begins at the spcified absolute time and ends at the end of the specified track.

preventremoval *()* -- Method on CD player object

Locks the eject button on the CD-ROM drive thus preventing the user from arbitrarily ejecting the caddy.

readda *(num_frames)* -- Method on CD player object

Reads the specified number of frames from an audio CD mounted in the CD-ROM drive. The return value is a string representing the audio frames. This string can be passed unaltered to the `parseframe` method of the parser object.

seek *(min, sec, frame)* -- Method on CD player object

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to an absolute time code location specified in minutes, seconds, and frames. The return value is the logical block number to which the pointer has been set.

seekblock *(block)* -- Method on CD player object

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified logical block number. The return value is the logical block number to which the pointer has been set.

seektrack *(track)* -- Method on CD player object

Sets the pointer that indicates the starting point of the next read of digital audio data from a CD-ROM. The pointer is set to the specified track. The return value is the logical block number to which the pointer has been set.

stop *()* -- Method on CD player object

Stops the current playing operation.

togglepause *()* -- Method on CD player object

Pauses the CD if it is playing, and makes it play if it is paused.

Parser objects (returned by `cd.createparser()`) have the following methods:

addcallback *(type, func, arg)* -- Method on CD parser object

Adds a callback for the parser. The parser has callbacks for eight different types of data in the digital audio data stream. Constants for these types are defined at the `cd` module level (see above). The callback is called as follows: `func(arg, type, data)`, where `arg` is the user supplied argument, `type` is the particular type of callback, and `data` is the data returned for this `type` of callback. The type of the data depends on the `type` of callback as follows:

`cd.audio:`

The argument is a string which can be passed unmodified to `al.writesamps()`.

`cd.pnum:`

The argument is an integer giving the program (track) number.

`cd.index:`

The argument is an integer giving the index number.

`cd.ptime:`

The argument is a tuple consisting of the program time in minutes, seconds, and frames.

`cd.atime:`

The argument is a tuple consisting of the absolute time in minutes, seconds, and frames.

`cd.catalog:`

The argument is a string of 13 characters, giving the catalog number of the CD.

`cd.ident:`

The argument is a string of 12 characters, giving the ISRC identification number of the recording. The string consists of two characters country code, three characters owner code, two characters giving the year, and five characters giving a serial number.

`cd.control:`

The argument is an integer giving the control bits from the CD subcode data.

deleteparser *()* -- Method on CD parser object

Deletes the parser and frees the memory it was using. The object should not be used after this call. This call is done automatically when the last reference to the object is removed.

parseframe *(frame)* -- Method on CD parser object

Parses one or more frames of digital audio data from a CD such as returned by `readda`. It determines which subcodes are present in the data. If these subcodes have changed since the last frame, then `parseframe` executes a callback of the appropriate type passing to it the subcode data found in the frame. Unlike the C function, more than one frame of digital audio data can be passed to this method.

removecallback *(type)* -- Method on CD parser object

Removes the callback for the given `type`.

resetparser *()* -- Method on CD parser object

Resets the fields of the parser used for tracking subcodes to an initial state. `resetparser` should be called after the disc has been changed.

Modules -- Python library reference

Next: Classes and Instances Prev: Other Built-in Types Up: Other Built-in Types Top: Top

2.1.7.1. 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'>.

Next: Classes and Instances Prev: Other Built-in Types Up: Other Built-in Types Top: Top

Top -- Python library reference

The Python library

This file 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. (These manuals are not yet available in INFO or Texinfo format.)

This version corresponds to Python version 1.3 (13 Oct 1995).

Menu

sunaudiodev -- Python library reference

17.1. Built-in Module sunaudiodev

This module allows you to access the sun audio interface. The sun audio hardware is capable of recording and playing back audio data in U-LAW format with a sample rate of 8K per second. A full description can be gotten with `` `man audio' ``.

The module defines the following variables and functions:

error -- exception of module sunaudiodev

> This exception is raised on all errors. The argument is a string describing what went wrong.

open *(mode)* -- function of module sunaudiodev

> 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.

Menu

__builtin__ -- Python library reference

3.10. Built-in Module __builtin__

This module provides direct access to all `built-in' identifiers of Python; e.g. `__builtin__.open` is the full name for the built-in function `open`. See the section on Built-in Functions in the previous chapter.

mactcp -- Python library reference

14.9. Built-in Module mactcp

This module provides an interface to the Macintosh TCP/IP driver MacTCP.`macdnr` which provides an interface to the name-server (allowing you to translate hostnames to ip-addresses), a module `MACTCP` which has symbolic names for constants constants used by MacTCP and a wrapper module `socket` which mimics the UNIX socket interface (as far as possible). It may not be available in all Mac Python versions.

A complete description of the MacTCP interface can be found in the Apple MacTCP API documentation.

MTU *()* -- function of module mactcp

> Return the Maximum Transmit Unit (the packet size) of the network interface.

IPAddr *()* -- function of module mactcp

> Return the 32-bit integer IP address of the network interface.

NetMask *()* -- function of module mactcp

> Return the 32-bit integer network mask of the interface.

TCPCreate *(size)* -- function of module mactcp

> Create a TCP Stream object. *size* is the size of the receive buffer, `4096` is suggested by various sources.

UDPCreate *(size, port)* -- function of module mactcp

> Create a UDP stream object. *size* is the size of the receive buffer (and, hence, the size of the biggest datagram you can receive on this port). *port* is the UDP port number you want to receive datagrams on, a value of zero will make MacTCP select a free port.

Menu

time -- Python library reference

6.2. Built-in Module time

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

An explanation of some terminology and conventions is in order.

> The ``epoch'' is the point where the time starts. On January 1st of that year, at 0 hours, the ``time since the epoch'' is zero. For UNIX, the epoch is 1970. To find out what the epoch is, look at `gmtime(0)`.

UTC is Coordinated Universal Time (formerly known as Greenwich Mean Time). The acronym UTC is not a mistake but a compromise between English and French.

DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most UNIX systems, the clock ``ticks'' only 50 or 100 times a second, and on the Mac, times are only accurate to whole seconds.

The module defines the following functions and data items:

altzone -- data of module time

> The offset of the local DST timezone, in seconds west of the 0th meridian, if one is defined. Negative if the local DST timezone is east of the 0th meridian (as in Western Europe, including the UK). Only use this if `daylight` is nonzero.

asctime *(tuple)* -- function of module time

> Convert a tuple representing a time as returned by `gmtime()` or `localtime()` to a 24-character string of the following form: `'Sun Jun 20 23:21:05 1993'`. Note: unlike the C function of the same name, there is no trailing newline.

clock *()* -- function of module time

> Return the current CPU time as a floating point number expressed in seconds. The precision, and in fact the very definiton of the meaning of ``CPU time'', depends on that of the C function of the same name.

ctime *(secs)* -- function of module time

> Convert a time expressed in seconds since the epoch to a string representing local time. `ctime(t)` is equivalent to `asctime(localtime(t))`.

daylight -- data of module time

> Nonzero if a DST timezone is defined.

gmtime *(secs)* -- function of module time

> 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)* -- function of module time

Like `gmtime` but converts to local time. The dst flag is set to 1 when DST applies to the given time.

mktime *(tuple)* -- function of module time

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)* -- function of module time

Suspend execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.

strftime *(format, tuple)* -- function of module time

Convert a tuple representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the format argument. See the `strftime(3)` man page for details of the syntax of format strings.

time *()* -- function of module time

Return the time as a floating point number expressed in seconds since the epoch, in UTC. Note that even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second.

timezone -- data of module time

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 -- data of module time

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.

8.5. Built-in Module dbm

The `dbm` module provides an interface to the UNIX `(n)dbm` library. Dbm objects behave like mappings (dictionaries), except that keys and values are always strings. Printing a dbm object doesn't print the keys and values, and the `items()` and `values()` methods are not supported.

See also the `gdbm` module, which provides a similar interface using the GNU GDBM library. The module defines the following constant and functions:

error -- exception of module dbm

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

open *(filename, [flag, [mode]])* -- function of module dbm

> Open a dbm database and return a dbm object. The *filename* argument is the name of the database file (without the .dir or .pag extensions).

The optional *flag* argument can be `'r'` (to open an existing database for reading only --- default), `'w'` (to open an existing database for reading and writing), `'c'` (which creates the database if it doesn't exist), or `'n'` (which always creates a new empty database).

The optional *mode* argument is the UNIX mode of the file, used only when the database has to be created. It defaults to octal `0666`.

getopt -- Python library reference

6.3. Standard Module getopt

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

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

```
>>> import getopt, string

>>> args = string.split('-a -b -cfoo -d bar 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.
Next: tempfile Prev: time Up: Generic Operating System Services Top: Top

MacOS -- Python library reference

14.7. Built-in Module MacOS

This module provides access to MacOS specific functionality in the python interpreter, such as how the interpreter eventloop functions and the like. Use with care.

Note the capitalisation of the module name, this is a historical artefact.

Error -- exception of module MacOS

> This exception is raised on MacOS generated errors, either from functions in this module or from other mac-specific modules like the toolbox interfaces. The arguments are the integer error code (the *OSErr* value) and a textual description of the error code.

SetHighLevelEventHandler *(handler)* -- function of module MacOS

> Pass a python function that will be called upon reception of a high-level event. The previous handler is returned. The handler function is called with the event as argument.

Note that your event handler is currently only called dependably if your main event loop is in *stdwin*.

AcceptHighLevelEvent *()* -- function of module MacOS

> Read a high-level event. The return value is a tuple `(sender, refcon, data)`.

SetScheduleTimes *(fgi, fgy [, bgi, bgy])* -- function of module MacOS

> Controls how often the interpreter checks the event queue and how long it will yield the processor to other processes. *fgi* specifies after how many clicks (one click is one 60th of a second) the interpreter should check the event queue, and *fgy* specifies for how many clicks the CPU should be yielded when in the foreground. The optional *bgi* and *bgy* allow you to specify different values to use when python runs in the background, otherwise the background values will be set the the same as the foreground values. The function returns nothing.

The default values, which are based on nothing at all, are 12, 6, 1 and 12 respectively.

EnableAppswitch *(onoff)* -- function of module MacOS

> Enable or disable the python event loop, based on the value of *onoff*. The old value is returned. If the event loop is disabled no time is granted to other applications, checking for command-period is not performed and it is impossible to switch applications. This should only be used by programs providing their own complete event loop.

Note that based on the compiler used to build python it is still possible to loose events even with the python event loop disabled. If you have a `sys.stdout` window its handler will often also look in the event queue. Making sure nothing is ever printed works around this.

HandleEvent *(ev)* -- function of module MacOS

> Pass the event record `ev` back to the python event loop, or possibly to the handler for the `sys.stdout` window (based on the compiler used to build python). This allows python programs that do their own event handling to still have some command-period and window-switching capability.

GetErrorString *(errno)* -- function of module MacOS

> Return the textual description of MacOS error code *errno*.

Function Index -- Python library reference

Function Index

Python Services -- Python library reference

Next: String Services Prev: Built-in Objects Up: Top Top: Top

3. Python Services

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

sys

--- Access system specific parameters and functions.

types

--- Names for all built-in types.

traceback

--- Print or retrieve a stack traceback.

pickle

--- Convert Python objects to streams of bytes and back.

shelve

--- Python object persistency.

copy

--- Shallow and deep copy operations.

marshal

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

imp

--- Access the implementation of the `import` statement.

__builtin__

--- The set of built-in functions.

__main__

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

Menu

__main__

fl -- Python library reference

16.4. Built-in Module fl

This module provides an interface to the FORMS Library by Mark Overmars. The source for the library can be retrieved by anonymous ftp from host `ftp.cs.ruu.nl', directory SGI/FORMS. It was last tested with version 2.0b.

Most functions are literal translations of their C equivalents, dropping the initial `fl_' from their name. Constants used by the library are defined in module `FL` described below.

The creation of objects is a little different in Python than in C: instead of the `current form' maintained by the library to which new FORMS objects are added, all functions that add a FORMS object to a form are methods of the Python object representing the form. Consequently, there are no Python equivalents for the C functions `fl_addto_form` and `fl_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 available, 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()`.

Menu

Comparisons -- Python library reference

Next: Numeric Types Prev: Boolean Operations Up: Types Top: Top

2.1.3. Comparisons

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

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).

Next: Numeric Types Prev: Boolean Operations Up: Types Top: Top

STDWIN Example -- Python library reference

15.1.7. Example

Here is a minimal example of using STDWIN in Python. It creates a window and draws the string ``Hello world'' in the top left corner of the window. The window will be correctly redrawn when covered and re-exposed. The program quits when the close icon or menu item is requested.

```
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()

FORMS Objects -- Python library reference

Prev: Form Objects Up: fl Top: Top

16.4.3. FORMS Objects

Besides methods specific to particular kinds of FORMS objects, all FORMS objects also have the following methods:

set_call_back *(function, argument)* -- Method on FORMS object

> 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 *()* -- Method on FORMS object

> Delete the object.

show_object *()* -- Method on FORMS object

> Show the object.

hide_object *()* -- Method on FORMS object

> Hide the object.

redraw_object *()* -- Method on FORMS object

> Redraw the object.

freeze_object *()* -- Method on FORMS object

> Freeze the object.

unfreeze_object *()* -- Method on FORMS 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)

Prev: Form Objects Up: fl Top: Top

__main__ -- Python library reference

3.11. 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.

The Python Profiler -- Python library reference

10. The Python Profiler

Written by James Roskind(1)

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

Menu

---------- Footnotes ----------

(1) Updated and converted to LaTeX by Guido van Rossum. The references to the old profiler are left in the text, although it no longer exists.

md5 -- Python library reference

13.1. Built-in Module md5

This module implements the interface to RSA's MD5 message digest algorithm (see also Internet RFC 1321). Its use is quite straightforward: use the `md5.new()` to create an md5 object. You can now feed this object with arbitrary strings using the `update()` method, and at any point you can ask it for the digest (a strong kind of 128-bit checksum, a.k.a. ``fingerprint'') of the contatenation of the strings fed to it so far using the `digest()` method.

For example, to obtain the digest of the string `"Nobody inspects the spammish repetition"`:

```
>>> import md5

>>> m = md5.new()

>>> m.update("Nobody inspects")

>>> m.update(" the spammish repetition")

>>> m.digest()

'\273d\234\203\335\036\245\311\331\336\311\241\215\360\377\351'
```

More condensed:

```
>>> md5.new("Nobody inspects the spammish repetition").digest()

'\273d\234\203\335\036\245\311\331\336\311\241\215\360\377\351'
```

new *([arg])* -- function of module md5

Return a new md5 object. If *arg* is present, the method call `update(arg)` is made.

md5 *([arg])* -- function of module md5

For backward compatibility reasons, this is an alternative name for the `new()` function.

An md5 object has the following methods:

update *(arg)* -- Method on md5

Update the md5 object with the string *arg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments, i.e. `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

digest *()* -- Method on md5

Return the digest of the strings passed to the `update()` method so far. This is an 8-byte string which may contain non-ASCII characters, including null bytes.

copy *()* -- Method on md5

Return a copy (``clone'') of the md5 object. This can be used to efficiently compute the digests of strings that share a common initial substring.

Next: mpz Prev: Cryptographic Services Up: Cryptographic Services Top: Top

AST Objects -- Python library reference

3.9.3. AST Objects

AST objects (returned by `expr()`, `suite()`, and `tuple2ast()`, described above) have no methods of their own. Some of the functions defined which accept an AST object as their first argument may change to object methods in the future.

Ordered and equality comparisons are supported between AST objects.

Internet and WWW -- Python library reference

11. Internet and WWW Services

The modules described in this chapter provide various services to World-Wide Web (WWW) clients and/or services, and a few modules related to news and email. They are all implemented in Python. Some of these modules require the presence of the system-dependent module `sockets`, which is currently only fully supported on Unix and Windows NT. Here is an overview:

cgi

--- Common Gateway Interface, used to interpret forms in server-side scripts.

urllib

--- Open an arbitrary object given by URL (requires sockets).

httplib

--- HTTP protocol client (requires sockets).

ftplib

--- FTP protocol client (requires sockets).

gopherlib

--- Gopher protocol client (requires sockets).

nntplib

--- NNTP protocol client (requires sockets).

urlparse

--- Parse a URL string into a tuple (addressing scheme identifier, network location, path, parameters, query string, fragment identifier).

htmllib

--- A (slow) parser for HTML files.

sgmllib

--- Only as much of an SGML parser as needed to parse HTML.

rfc822

--- Parse RFC-822 style mail headers.

mimetools

--- Tools for parsing MIME style message bodies.

Menu

Next: Multimedia Services Prev: The Python Profiler Up: Top Top: Top

UDP Stream Objects -- Python library reference

Prev: TCP Status Objects Up: mactcp Top: Top

14.9.3. UDP Stream Objects

Note that, unlike the name suggests, there is nothing stream-like about UDP.

asr -- attribute of UDP stream

> The asynchronous service routine to be called on events such as datagram arrival without outstanding `Read` call. The `asr` has a single argument, the event code.

port -- attribute of UDP stream

> A read-only member giving the port number of this UDP stream.

Read *(timeout)* -- Method on UDP stream

> Read a datagram, waiting at most *timeout* seconds (is infinite). Return the data.

Write *(host, port, buf)* -- Method on UDP stream

> Send *buf* as a datagram to IP-address *host*, port *port*.

Prev: TCP Status Objects Up: mactcp Top: Top

Introduction -- Python library reference

Next: Built-in Objects Prev: Top Up: Top Top: Top

1. Introduction

The ``Python library'' contains several different kinds of components.

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

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

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

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

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

Let the show begin!

Next: Built-in Objects Prev: Top Up: Top Top: Top

Menu Objects -- Python library reference

15.1.4. Menu Objects

A menu object represents a menu. The menu is destroyed when the menu object is deleted. The following methods are defined:

additem *(text, shortcut)* -- Method on menu

> 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)* -- Method on menu

> Set the text of item number *i*.

enable *(i, flag)* -- Method on menu

> Enable or disables item *i*.

check *(i, flag)* -- Method on menu

> Set or clear the check mark for item *i*.

close *()* -- Method on menu

> Discard the menu object. It should not be used again.

Next: Limitations Prev: Deterministic Profiling Up: The Python Profiler Top: Top

10.5. Reference Manual

The primary entry point for the profiler is the global function `profile.run()`. It is typically used to create any profile information. The reports are formatted and printed using methods of the class `pstats.Stats`. The following is a description of all of these standard entry points and functions. For a more in-depth view of some of the code, consider reading the later section on Profiler Extensions, which includes discussion of how to derive ``better'' profilers from the classes presented, or reading the source code for these modules.

profile.run *(string[, filename[, ...]])* -- profiler function

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

2706 function calls (2004 primitive calls) in 4.504 CPU seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall filename:lineno(function)
2	0.006	0.003	0.953	0.477 pobject.py:75(save_objects)
43/3	0.533	0.012	0.749	0.250 pobject.py:99(evaluate)
...				

}

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

ncalls

for the number of calls,

tottime

for the total time spent in the given function (and excluding time made in calls to sub-functions),

percall

is the quotient of `tottime` divided by `ncalls`

cumtime

is the total time spent in this and all subfunctions (i.e., from invocation till exit). This figure is accurate *even* for recursive functions.

percall

is the quotient of `cumtime` divided by primitive calls

filename:lineno(function)

provides the respective data of each function

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

pstats.Stats *(filename[, ...])* -- profiler function

This class constructor creates an instance of a ``statistics object'' from a *filename* (or set of filenames). `Stats` objects are manipulated by methods, in order to print useful reports.

The file selected by the above constructor must have been created by the corresponding version of `profile`. To be specific, there is *NO* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers (e.g., the old system profiler).

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

Menu

imp -- Python library reference

3.8. Built-in Module imp

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

get_magic *()* -- function of module imp

> Return the magic string value used to recognize byte-compiled code files (``.pyc` files'').

get_suffixes *()* -- function of module imp

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

find_module *(name, [path])* -- function of module imp

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

init_builtin *(name)* -- function of module imp

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

init_frozen *(name)* -- function of module imp

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

is_builtin *(name)* -- function of module imp

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

is_frozen *(name)* -- function of module imp

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

load_compiled *(name, pathname, [file])* -- function of module imp

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

user-defined class emulating a file.

load_dynamic *(name, pathname, [file])* -- function of module imp

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

load_source *(name, pathname, [file])* -- function of module imp

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

new_module *(name)* -- function of module imp

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

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

SEARCH_ERROR -- data of module imp

The module was not found.

PY_SOURCE -- data of module imp

The module was found as a source file.

PY_COMPILED -- data of module imp

The module was found as a compiled code object file.

C_EXTENSION -- data of module imp

The module was found as dynamically loadable shared library.

Menu

Examples

urllib -- Python library reference

11.2. Standard Module urllib

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

it defines the following public functions:

urlopen *(url)* -- function of module urllib

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

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

urlretrieve *(url)* -- function of module urllib

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

urlcleanup *()* -- function of module urllib

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

quote *(string[, addsafe])* -- function of module urllib

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

Example: `quote('/~conolly/')` yields `'/%7econnolly/'`.

unquote *(string)* -- function of module urllib

> Replace `` `%xx' `` escapes by their single-character equivalent.

Example: `unquote('/%7Econnolly/')` yields `'/~connolly/'`.

Restrictions:

> Currently, only the following protocols are supported: HTTP, (versions 0.9 and 1.0), Gopher (but not Gopher-+), FTP, and local files.

The caching feature of `urlretrieve()` has been disabled until I find the time to hack proper processing of Expiration time headers.

There should be a function to query whether a particular URL is in the cache.

For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.

The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.

The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (e.g. an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the `Content-type` header. For the Gopher protocol, type information is encoded in the URL; there is currently no easy way to extract it. If the returned data is HTML, you can use the module `htmllib` to parse it.

Although the `urllib` module contains (undocumented) routines to parse and unparse URL strings, the recommended interface for URL manipulation is in module `urlparse`.

FSSpec objects -- Python library reference

14.6.1. FSSpec objects

data -- attribute of FSSpec object

The raw data from the FSSpec object, suitable for passing to other applications, for instance.

as_pathname *()* -- Method on FSSpec object

Return the full pathname of the file described by the FSSpec object.

as_tuple *()* -- Method on FSSpec object

Return the `(wdRefNum, parID, name)` tuple of the file described by the FSSpec object.

NewAlias *([file])* -- Method on FSSpec object

Create an Alias object pointing to the file described by this FSSpec. If the optional *file* parameter is present the alias will be relative to that file, otherwise it will be absolute.

NewAliasMinimal *()* -- Method on FSSpec object

Create a minimal alias pointing to this file.

GetCreatorType *()* -- Method on FSSpec object

Return the 4-char creator and type of the file.

SetCreatorType *(creator, type)* -- Method on FSSpec object

Set the 4-char creator and type of the file.

GetFInfo *()* -- Method on FSSpec object

Return a FInfo object describing the finder info for the file.

SetFInfo *(finfo)* -- Method on FSSpec object

Set the finder info for the file to the values specified in the *finfo* object.

HTTP Example -- Python library reference

11.3.2. Example

Here is an example session:

```
>>> import httplib

>>> h = httplib.HTTP('www.cwi.nl')

>>> h.putrequest('GET', '/index.html')

>>> h.putheader('Accept', 'text/html')

>>> h.putheader('Accept', 'text/plain')

>>> h.endheaders()

>>> errcode, errmsg, headers = h.getreply()

>>> print errcode # Should be 200

>>> f = h.getfile()

>>> data f.read() # Get the raw HTML

>>> f.close()

>>>
```

mpz -- Python library reference

13.2. Built-in Module mpz

This is an optional module. It is only available when Python is configured to include it, which requires that the GNU MP software is installed.

This module implements the interface to part of the GNU MP library, which defines 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 exact description). An mpz-number is printed like this: `mpz(`*`value`*`)`.

mpz *(value)* -- function of module mpz

> 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)* -- function of module mpz

> 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)* -- function of module mpz

> Return the greatest common divisor of *op1* and *op2*.

gcdext *(a, b)* -- function of module mpz

> Return a tuple *`(g, s, t)`*, such that *`a*s + b*t == g == gcd(a, b)`*.

sqrt *(op)* -- function of module mpz

> Return the square root of *op*. The result is rounded towards zero.

sqrtrem *(op)* -- function of module mpz

> Return a tuple *`(root, remainder)`*, such that *`root*root + remainder == op`*.

divm *(numerator, denominator, modulus)* -- function of module mpz

> 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 *()* -- Method on mpz

> 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.

Built-in Objects -- Python library reference

2. Built-in Types, Exceptions and Functions

Names for built-in exceptions and functions are found in a separate symbol table. This table is searched last when the interpreter looks up the meaning of a name, so local and global user-defined names can override built-in names. Built-in types are described together here for easy reference.(1) The tables in this chapter document the priorities of operators by listing them in order of ascending priority (within a table) and grouping operators that have the same priority in the same box. Binary operators of the same priority group from left to right. (Unary operators group from right to left, but there you have no real choice.) See Chapter 5 of the Python Reference Manual for the complete picture on operator priorities.

Menu

---------- Footnotes ----------

(1) Most descriptions sorely lack explanations of the exceptions that may be raised --- this will be fixed in a future version of this manual.

imgfile -- Python library reference

16.10. Built-in Module imgfile

The imgfile module allows python programs to access SGI imglib image files (also known as .rgb files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

error -- exception of module imgfile

> This exception is raised on all errors, such as unsupported file type, etc.

getsizes *(file)* -- function of module imgfile

> 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)* -- function of module imgfile

> 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])* -- function of module imgfile

> 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)* -- function of module imgfile

> 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)* -- function of module imgfile

> 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`.

syslog -- Python library reference

8.11. Built-in Module syslog

This module provides an interface to the Unix `syslog` library routines. Refer to the UNIX manual pages for a detailed description of the `syslog` facility.

The module defines the following functions:

syslog *([priority,] message)* -- Method on posixfile

> Send the string *message* to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a *facility* and a *level*. The optional *priority* argument, which defaults to `(LOG_USER | LOG_INFO)`, determines the message priority.

openlog *(ident, [logopt, [facility]])* -- Method on posixfile

> Logging options other than the defaults can be set by explicitly opening the log file with `openlog()` prior to calling `syslog()`. The defaults are (usually) *ident* = `` `syslog ``', *logopt* = 0, *facility* = `LOG_USER`. The *ident* argument is a string which is prepended to every message. The optional *logopt* argument is a bit field - see below for possible values to combine. The optional *facility* argument sets the default facility for messages which do not have a facility explicitly encoded.

closelog *()* -- Method on posixfile

> Close the log file.

setlogmask *(maskpri)* -- Method on posixfile

> This function set the priority mask to *maskpri* and returns the previous mask value. Calls to `syslog` with a priority level not set in *maskpri* are ignored. The default is to log all priorities. The function `LOG_MASK(`*`pri`*`)` calculates the mask for the individual priority *pri*. The function `LOG_UPTO(`*`pri`*`)` calculates the mask for all priorities up to and including *pri*.

The module defines the following constants:

Priority levels (high to low):

> `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

Facilities:

> `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON` and `LOG_LOCAL0` to `LOG_LOCAL7`.

Log options:

> `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`, `LOG_NOWAIT` and `LOG_PERROR` if defined in syslog.h.

Limitations -- Python library reference

Next: Calibration Prev: Reference Manual Up: The Python Profiler Top: Top

10.6. Limitations

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

The second limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying ``clock'' is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate that that underlying clock. If enough measurements are taken, then the ``error'' will tend to average out. Unfortunately, removing this first error induces a second source of error...

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

Next: Calibration Prev: Reference Manual Up: The Python Profiler Top: Top

select -- Python library reference

7.3. Built-in Module select

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

error -- exception of module select

> 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])* -- function of module select

> 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 *timeout* argument specifies a time-out 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).

rotor -- Python library reference

13.3. Built-in Module rotor

This module implements a rotor-based encryption algorithm, contributed by Lance Ellinghouse. The design is derived from the Enigma device, a machine used during World War II to encipher messages. A rotor is simply a permutation. For example, if the character `A' is the origin of the rotor, then a given rotor might map `A' to `L', `B' to `Z', `C' to `G', and so on. To encrypt, we choose several different rotors, and set the origins of the rotors to known positions; their initial position is the ciphering key. To encipher a character, we permute the original character by the first rotor, and then apply the second rotor's permutation to the result. We continue until we've applied all the rotors; the resulting character is our ciphertext. We then change the origin of the final rotor by one position, from `A' to `B'; if the final rotor has made a complete revolution, then we rotate the next-to-last rotor by one position, and apply the same procedure recursively. In other words, after enciphering one character, we advance the rotors in the same fashion as a car's odometer. Decoding works in the same way, except we reverse the permutations and apply them in the opposite order. The available functions in this module are:

newrotor *(key[, numrotors])* -- function of module rotor

> Return a rotor object. *key* is a string containing the encryption key for the object; it can contain arbitrary binary data. The key will be used to randomly generate the rotor permutations and their initial positions. *numrotors* is the number of rotor permutations in the returned object; if it is omitted, a default value of 6 will be used.

Rotor objects have the following methods:

setkey *()* -- Method on rotor

> Reset the rotor to its initial state.

encrypt *(plaintext)* -- Method on rotor

> Reset the rotor object to its initial state and encrypt *plaintext*, returning a string containing the ciphertext. The ciphertext is always the same length as the original plaintext.

encryptmore *(plaintext)* -- Method on rotor

> Encrypt *plaintext* without resetting the rotor object, and return a string containing the ciphertext.

decrypt *(ciphertext)* -- Method on rotor

> Reset the rotor object to its initial state and decrypt *ciphertext*, returning a string containing the ciphertext. The plaintext string will always be the same length as the ciphertext.

decryptmore *(ciphertext)* -- Method on rotor

> Decrypt *ciphertext* without resetting the rotor object, and return a string containing the ciphertext.

An example usage:

```
>>> import rotor
```

>>> rt = rotor.newrotor('key', 12)

>>> rt.encrypt('bar')

'\2534\363'

```
>>> rt.encryptmore('bar')

'\357\375$'

>>> rt.encrypt('bar')

'\2534\363'

>>> rt.decrypt('\2534\363')

'bar'

>>> rt.decryptmore('\357\375$')

'bar'

>>> rt.decrypt('\357\375$')

'l(\315'

>>> del rt
```

The module's code is not an exact simulation of the original Enigma device; it implements the rotor encryption scheme differently from the original. The most important difference is that in the original Enigma, there were only 5 or 6 different rotors in existence, and they were applied twice to each character; the cipher key was the order in which they were placed in the machine. The Python rotor module uses the supplied key to initialize a random number generator; the rotor permutations and their initial positions are then randomly generated. The original device only enciphered the letters of the alphabet, while this module can handle any 8-bit binary data; it also produces binary output. This module can also operate with an arbitrary number of rotors.

The original Enigma cipher was broken in 1944. The version implemented here is probably a good deal more difficult to crack (especially if you use many rotors), but it won't be impossible for a truly skilful and determined attacker to break the cipher. So if you want to keep the NSA out of your files, this rotor cipher may well be unsafe, but for discouraging casual snooping through your files, it will probably be just fine, and may be somewhat safer than using the Unix crypt command.

Message Objects -- Python library reference

11.10.1. Message Objects

A `Message` instance has the following methods:

rewindbody *()* -- function of module rfc822

> Seek to the start of the message body. This only works if the file object is seekable.

getallmatchingheaders *(name)* -- function of module rfc822

> Return a list of lines consisting of all headers matching *name*, if any. Each physical line, whether it is a continuation line or not, is a separate list item. Return the empty list if no header matches *name*.

getfirstmatchingheader *(name)* -- function of module rfc822

> Return a list of lines comprising the first header matching *name*, and its continuation line(s), if any. Return `None` if there is no header matching *name*.

getrawheader *(name)* -- function of module rfc822

> Return a single string consisting of the text after the colon in the first header matching *name*. This includes leading whitespace, the trailing linefeed, and internal linefeeds and whitespace if there any continuation line(s) were present. Return `None` if there is no header matching *name*.

getheader *(name)* -- function of module rfc822

> Like `getrawheader(name)`, but strip leading and trailing whitespace (but not internal whitespace).

getaddr *(name)* -- function of module rfc822

> Return a pair (full name, email address) parsed from the string returned by `getheader(name)`. If no header matching *name* exists, return `None, None`; otherwise both the full name and the address are (possibly empty)strings.

Example: If `m`'s first `From` header contains the string* `'jack@cwi.nl (Jack Jansen)'`, then `m.getaddr('From')` will yield the pair `('Jack Jansen', 'jack@cwi.nl')`. If the header contained `'Jack Jansen <jack@cwi.nl>'` instead, it would yield the exact same result.

getaddrlist *(name)* -- function of module rfc822

> This is similar to `getaddr(list)`, but parses a header containing a list of email addresses (e.g. a `To` header) and returns a list of (full name, email address) pairs (even if there was only one address in the header). If there is no header matching *name*, return an empty list.

XXX The current version of this function is not really correct. It yields bogus results if a full name contains a comma.

getdate *(name)* -- function of module rfc822

> Retrieve a header using `getheader` and parse it into a 9-tuple compatible with `time.mktime()`. If there is no header matching *name*, or it is unparsable, return `None`.

Date parsing appears to be a black art, and not all mailers adhere to the standard. While it has been tested and found correct on a large collection of email from many sources, it is still possible that this function may occasionally yield an incorrect result.

`Message` instances also support a read-only mapping interface. In particular: `m[name]` is the same as `m.getheader(name)`; and `len(m)`, `m.has_key(name)`, `m.keys()`, `m.values()` and `m.items()` act as expected (and consistently).

Finally, `Message` instances have two public instance variables:

headers -- data of module rfc822

A list containing the entire set of header lines, in the order in which they were read. Each line contains a trailing newline. The blank line terminating the headers is not contained in the list.

fp -- data of module rfc822

The file object passed at instantiation time.

FL (uppercase) -- Python library reference

16.5. Standard Module FL

This module defines symbolic constants needed to use the built-in module `fl` (see above); they are equivalent to those defined in the C header file <forms.h> except that the name prefix ``FL_`' is omitted. Read the module source for a complete list of the defined names. Suggested use:

```
import fl
```

from FL import *

signal -- Python library reference

7.1. Built-in Module signal

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

> A handler for a particular signal, once set, remains installed until it is explicitly reset (i.e. Python uses the BSD style interface).

There is no way to ``block'' signals temporarily from critical sections (since this is not supported by all UNIX flavors).

Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the ``atomic'' instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (e.g. regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.

When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying UNIX system's semantics regarding interrupted system calls.

Because the C signal handler always returns, it makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV`.

Python installs a small number of signal handlers by default: `SIGPIPE` is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions), `SIGINT` is translated into a `KeyboardInterrupt` exception, and `SIGTERM` is caught so that necessary cleanup (especially `sys.exitfunc`) can be performed before actually terminating. All of these can be overridden.

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

The variables defined in the signal module are:

SIG_DFL -- data of module signal

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

SIG_IGN -- data of module signal

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

SIG* -- data of module signal

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

NSIG -- data of module signal

One more than the number of the highest signal number.

The signal module defines the following functions:

alarm *(time)* -- function of module signal

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

getsignal *(signalnum)* -- function of module signal

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

pause *()* -- function of module signal

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

signal *(signalnum, handler)* -- function of module signal

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

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

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

GL and DEVICE -- Python library reference

16.9. Standard Modules GL and DEVICE

These modules define the constants used by the Silicon Graphics *Graphics Library* that C programmers find in the header files <gl/gl.h> and <gl/device.h>. Read the module source files for details.

htmllib -- Python library reference

11.8. Standard Module htmllib

This module defines a number of classes which can serve as a basis for parsing text files formatted in HTML (HyperText Mark-up Language). The classes are not directly concerned with I/O --- the have to be fed their input in string form, and will make calls to methods of a ``formatter'' object in order to produce output. The classes are designed to be used as base classes for other classes in order to add functionality, and allow most of their methods to be extended or overridden. In turn, the classes are derived from and extend the class `SGMLParser` defined in module `sgmllib`. The following is a summary of the interface defined by `sgmllib.SGMLParser`:

> The interface to feed data to an instance is through the `feed()` method, which takes a string argument. This can be called with as little or as much text at a time as desired; `p.feed(a); p.feed(b)` has the same effect as `p.feed(a+b)`. When the data contains complete HTML elements, these are processed immediately; incomplete elements are saved in a buffer. To force processing of all unprocessed data, call the `close()` method.

Example: to parse the entire contents of a file, do* `parser.feed(open(file).read()); parser.close()`.

The interface to define semantics for HTML tags is very simple: derive a class and define methods called `start_`*`tag`*`()`, `end_`*`tag`*`()`, or `do_`*`tag`*`()`. The parser will call these at appropriate moments: `start_`*`tag`* or `do_`*`tag`* is called when an opening tag of the form `<`*`tag ...`*`>` is encountered; `end_`*`tag`* is called when a closing tag of the form `<`*`tag`*`>` is encountered. If an opening tag requires a corresponding closing tag, like `<H1>` ... `</H1>`, the class should define the `start_`*`tag`* method; if a tag requires no closing tag, like `<P>`, the class should define the `do_`*`tag`* method.

The module defines the following classes:

HTMLParser *()* -- function of module htmllib

> This is the most basic HTML parser class. It defines one additional entity name over the names defined by the `SGMLParser` base class, `•`. It also defines handlers for the following tags: `<LISTING>...</LISTING>`, `<XMP>...</XMP>`, and `<PLAINTEXT>` (the latter is terminated only by end of file).

CollectingParser *()* -- function of module htmllib

> This class, derived from `HTMLParser`, collects various useful bits of information from the HTML text. To this end it defines additional handlers for the following tags: `<A>...</A>`, `<HEAD>...</HEAD>`, `<BODY>...</BODY>`, `<TITLE>...</TITLE>`, `<NEXTID>`, and `<ISINDEX>`.

FormattingParser *(formatter, stylesheet)* -- function of module htmllib

> This class, derived from `CollectingParser`, interprets a wide selection of HTML tags so it can produce formatted output from the parsed data. It is initialized with two objects, a *formatter* which should define a number of methods to format text into paragraphs, and a *stylesheet* which defines a number of static parameters for the formatting process. Formatters and style sheets are documented later in this section.

AnchoringParser *(formatter, stylesheet)* -- function of module htmllib

> This class, derived from `FormattingParser`, extends the handling of the `<A>...</A>` tag pair to call the formatter's `bgn_anchor()` and `end_anchor()` methods. This allows the formatter to display the anchor in a different font or color, etc.

Instances of `CollectingParser` (and thus also instances of `FormattingParser` and `AnchoringParser`) have the following instance variables:

anchornames -- data of module htmllib

A list of the values of the `NAME` attributes of the `<A>` tags encountered.

anchors -- data of module htmllib

A list of the values of `HREF` attributes of the `<A>` tags encountered.

anchortypes -- data of module htmllib

A list of the values of the `TYPE` attributes of the `<A>` tags encountered.

inanchor -- data of module htmllib

Outside an `<A>...</A>` tag pair, this is zero. Inside such a pair, it is a unique integer, which is positive if the anchor has a `HREF` attribute, negative if it hasn't. Its absolute value is one more than the index of the anchor in the `anchors`, `anchornames` and `anchortypes` lists.

isindex -- data of module htmllib

True if the `<ISINDEX>` tag has been encountered.

nextid -- data of module htmllib

The attribute list of the last `<NEXTID>` tag encountered, or an empty list if none.

title -- data of module htmllib

The text inside the last `<TITLE>...</TITLE>` tag pair, or `''` if no title has been encountered yet.

The `anchors`, `anchornames` and `anchortypes` lists are ``parallel arrays'': items in these lists with the same index pertain to the same anchor. Missing attributes default to the empty string. Anchors with neither a `HREF` nor a `NAME` attribute are not entered in these lists at all.

The module also defines a number of style sheet classes. These should never be instantiated --- their class variables are the only behavior required. Note that style sheets are specifically designed for a particular formatter implementation. The currently defined style sheets are:

NullStylesheet -- data of module htmllib

A style sheet for use on a dumb output device such as an ASCII terminal.

X11Stylesheet -- data of module htmllib

A style sheet for use with an X11 server.

MacStylesheet -- data of module htmllib

A style sheet for use on Apple Macintosh computers.

StdwinStylesheet -- data of module htmllib

A style sheet for use with the `stdwin` module; it is an alias for either `X11Stylesheet` or `MacStylesheet`.

GLStylesheet -- data of module htmllib

A style sheet for use with the SGI Graphics Library and its font manager (the SGI-specific built-in modules `gl` and `fm`).

Style sheets have the following class variables:

stdfontset -- data of module htmllib

A list of up to four font definititions, respectively for the roman, italic, bold and constant-width variant of a font for normal text. If the list contains less than four font definitions, the last item is used as the default for missing items. The type of a font definition depends on the formatter in use; its only use is as a parameter to the formatter's `setfont()` method.

h1fontset -- data of module htmllib

h2fontset -- data of module htmllib

h3fontset -- data of module htmllib

The font set used for various headers (text inside `<H1>...</H1>` tag pairs etc.).

stdindent -- data of module htmllib

The indentation of normal text. This is measured in the ``native'' units of the formatter in use; for some formatters these are characters, for others (especially those that actually support variable-spacing fonts) in pixels or printer points.

ddindent -- data of module htmllib

The indentation used for the first level of `<DD>` tags.

ulindent -- data of module htmllib

The indentation used for the first level of `<UL>` tags.

h1indent -- data of module htmllib

The indentation used for level 1 headers.

h2indent -- data of module htmllib

The indentation used for level 2 headers.

literalindent -- data of module htmllib

The indentation used for literal text (text inside `<PRE>...</PRE>` and similar tag pairs).

Although no documented implementation of a formatter exists, the `FormattingParser` class assumes that formatters have a certain interface. This interface requires the following methods:

setfont *(fontspec)* -- function of module htmllib

Set the font to be used subsequently. The *fontspec* argument is an item in a style sheet's font set.

flush *()* -- function of module htmllib

Finish the current line, if not empty, and begin a new one.

setleftindent *(n)* -- function of module htmllib

Set the left indentation of the following lines to *n* units.

needvspace *(n)* -- function of module htmllib

Require at least *n* blank lines before the next line. Implies `flush()`.

addword *(word, space)* -- function of module htmllib

Add a *word* to the current paragraph, followed by *space* spaces.

nospace -- data of module htmllib

If this instance variable is true, empty words should be ignored by `addword`. It should be set to false after a non-empty word has been added.

setjust *(justification)* -- function of module htmllib

Set the justification of the current paragraph. The *justification* can be `'c'` (center), `'l'` (left justified), `'r'` (right justified) or `'lr'` (left and right justified).

bgn_anchor *(id)* -- function of module htmllib

Begin an anchor. The *id* parameter is the value of the parser's `inanchor` attribute.

end_anchor *(id)* -- function of module htmllib

End an anchor. The *id* parameter is the value of the parser's `inanchor` attribute.

A sample formatter implementation can be found in the module `fmt`, which in turn uses the module `Para`. These modules are not intended as standard library modules; they are available as an example of how to write a formatter.

Built-in Functions -- Python library reference

Prev: Exceptions Up: Built-in Objects Top: Top

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)* -- built-in function

> Return the absolute value of a number. The argument may be a plain or long integer or a floating point number.

apply *(function, args[, keywords])* -- built-in function

> 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.) If the optional *keywords* argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the the argument list.

chr *(i)* -- built-in function

> 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)* -- built-in function

> 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)* -- built-in function

> 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)* -- built-in function

> Compile the *string* into a code object. Code objects can be executed by an `exec` statement or evaluated by a call to `eval()`. The *filename* argument should give the file from which the code was read; pass e.g. `'<string>'` if it wasn't read from a file. The *kind* argument specifies what kind of code must be compiled; it can be `'exec'` if *string* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something else than `None` will printed).

delattr *(object, name)* -- built-in function

> This is a relative of `setattr`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`.

dir *()* -- built-in function

> 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)* -- built-in function

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 *(expression[, globals[, locals]])* -- built-in function

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

Hints: dynamic execution of statements is supported by the `exec` statement. Execution of statements from a file is supported by the `execfile()` function. The `globals()` and `locals()` functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `execfile()`.

execfile *(file[, globals[, locals]])* -- built-in function

This function is similar to the `exec` statement, but parses a file instead of a string. It is different from the `import` statement in that it does not use the module administration --- it reads the file unconditionally and does not create a new module.(1)

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the *globals* and *locals* dictionaries as global and local name space. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where `execfile()` is called. The return value is `None`.

filter *(function, list)* -- built-in function

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)* -- built-in function

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

getattr *(object, name)* -- built-in function

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`.

globals *()* -- built-in function

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

hasattr *(object, name)* -- built-in function

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)* -- built-in function

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)* -- built-in function

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

id *(object)* -- built-in function

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])* -- built-in function

Almost equivalent to `eval(raw_input(prompt))`. Like `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)* -- built-in function

Convert a number to a plain integer. The argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers is defined by the C semantics; normally the conversion truncates towards zero.(2)

len *(s)* -- built-in function

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

locals *()* -- built-in function

Return a dictionary representing the current local symbol table. Inside a function, modifying this dictionary does not always have the desired effect.

long *(x)* -- built-in function

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

map *(function, list, ...)* -- built-in function

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)* -- built-in function

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

min *(s)* -- built-in function

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

oct *(x)* -- built-in function

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

open *(filename[, mode[, bufsize]])* -- built-in function

Return a new file object (described earlier under Built-in Types). The first two arguments are the same as for `stdio`'s `fopen()`: *filename* is the file name to be opened, *mode* indicates how the file is to be opened: `'r'` for reading, `'w'` for writing (truncating an existing file), and `'a'` opens it for appending. Modes `'r+'`, `'w+'` and `'a+'` open the file for updating, provided the underlying `stdio` library understands this. On systems that differentiate between binary and text files, `'b'` appended to the mode opens the file in binary mode. If the file cannot be opened, `IOError` is raised. If *mode* is omitted, it defaults to `'r'`. The optional *bufsize* argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative *bufsize* means to use the system default, which is usually line buffered for for tty devices and fully buffered for other files.(3)

ord *(c)* -- built-in function

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[, z])* -- built-in function

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

range *([start,] end[, step])* -- built-in function

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

```
>>> 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])* -- built-in function

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

```
>>> s = raw_input('--> ')
```

--> Monty Python's Flying Circus

>>> s

"Monty Python's Flying Circus"

>>>

reduce *(function, list[, initializer])* -- built-in function

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)* -- built-in function

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. The return value is the module object (i.e. the same as the *module* argument).

There are a number of caveats:

If a module is syntactically correct but its initialization fails, the first `import` statement for it does not bind its name locally, but does store a (partially initialized) module object in `sys.modules`. To reload the module you must first `import` it again (this will bind the name to the partially initialized module object) before you can `reload()` it.

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

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

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

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

repr *(object)* -- built-in function

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)* -- built-in function

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)* -- built-in function

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)* -- built-in function

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

tuple *(sequence)* -- built-in function

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

type *(object)* -- built-in function

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

```
>>> import types
```

>>> if type(x) == types.StringType: print "It's a string"

vars *([object])* -- built-in function

Without arguments, return a dictionary corresponding to the current local symbol table. With a module, class or class instance object as argument (or anything else that has a `__dict__` attribute), returns a dictionary corresponding to the object's symbol table. The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined.(4)

xrange *([start,] end[, step])* -- built-in function

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

---------- Footnotes ----------

(1) It is used relatively rarely so does not warrant being made into a statement.

(2) This is ugly --- the language definition should require truncation towards zero.

(3) Specifying a buffer size currently has no effect on systems that don't have `setvbuf()`. The interface to specify the buffer size is not done using a method that calls `setvbuf()`, because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.

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

marshal -- Python library reference

3.7. Built-in Module marshal

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

This is not a general ``persistency'' module. For general persistency and transfer of Python objects through RPC calls, see the modules `pickle` and `shelve`. The `marshal` module exists mainly to support reading and writing the ``pseudo-compiled'' code for Python modules of `` `.pyc' `` files. Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported: `None`, integers, long integers, floating point numbers, strings, tuples, lists, dictionaries, and code objects, where it should be understood that tuples, lists and dictionaries are only supported as long as the values contained therein are themselves supported; and recursive lists and dictionaries should not be written (they will cause infinite loops).

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

There are functions that read/write files as well as functions operating on strings.

The module defines these functions:

dump *(value, file)* -- function of module marshal

> 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)* -- function of module marshal

> 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)* -- function of module marshal

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

loads *(string)* -- function of module marshal

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

---------- Footnotes ----------

(1) 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.

(2) A solution would be to refuse such literals in the parser, since they are inherently non-portable.

Another solution would be to let the `marshal` module raise an exception when an integer value would be truncated. At least one of these solutions will be implemented in a future version.

Next: imp Prev: copy Up: Python Services Top: Top

Debugger Commands -- Python library reference

9.1. Debugger Commands

The debugger recognizes the following commands. Most commands can be abbreviated to one or two letters; e.g. ``h(elp)" means that either ``h" or ``help" can be used to enter the help command (but not ``he" or ``hel", nor ``H" or ``Help or ``HELP"). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets (``[]") in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (``|").

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

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

h(elp) [*command*

]

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

w(here)

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

d(own)

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

u(p)

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

b(reak) [*lineno*|*function*

]

With a *lineno* argument, set a break there in the current file. With a *function* argument, set a break at the entry of that function. Without argument, list all breaks.

cl(ear) [*lineno*

]

With a *lineno* argument, clear that break in the current file. Without argument, clear all breaks (but first ask confirmation).

s(tep)

Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)

Continue execution until the next line in the current function is reached or it returns. (The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

r(eturn)

Continue execution until the current function returns.

c(ont(inue))

Continue execution, only stop when a breakpoint is encountered.

l(ist) [*first [, last*

]]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

a(rgs)

Print the argument list of the current function.

p *expression*

Evaluate the *expression* in the current context and print its value. (Note: `print` can also be used, but is not a debugger command --- this executes the Python `print` statement.)

[!

statement]

Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command. To set a global variable, you can prefix the assignment command with a ``global'' command on the same line, e.g.:

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

(Pdb)

q(uit)

Quit from the debugger. The program being executed is aborted.

tempfile -- Python library reference

6.4. Standard Module tempfile

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

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

The module defines a single user-callable function:

mktemp *()* -- function of module tempfile

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

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

tempdir -- data of module tempfile

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

template -- data of module tempfile

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

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

Application objects -- Python library reference

14.12.1. Application objects

Application objects have the following methods, among others:

makeusermenus *()* -- Method on Application

> Override this method if you need menus in your application. Append the menus to `self.menubar`.

getabouttext *()* -- Method on Application

> Override this method to return a text string describing your application. Alternatively, override the `do_about` method for more elaborate about messages.

mainloop *([mask, wait])* -- Method on Application

> This routine is the main event loop, call it to set your application rolling. *Mask* is the mask of events you want to handle, *wait* is the number of ticks you want to leave to other concurrent application (default 0, which is probably not a good idea). This method does not return until `self` is raised.

The event loop is split into many small parts, each of which can be overridden. The default methods take care of dispatching events to windows and dialogs, handling drags and resizes, Apple Events, events for non-FrameWork windows, etc.

do_char *(c, event)* -- Method on Application

> The user typed character *c*. The complete details of the event can be found in the *event* structure. This method can also be provided in a `Window` object, which overrides the application-wide handler if the window is frontmost.

do_dialogevent *(event)* -- Method on Application

> Called early in the event loop to handle modeless dialog events. The default method simply dispatches the event to the relevant dialog (not through the the `DialogWindow` object involved). Override if you need special handling of dialog events (keyboard shortcuts, etc).

connection object -- Python library reference

14.3.1. connection object

For all connection methods that take a *timeout* argument, a value of `-1` is indefinite, meaning that the command runs to completion.

callback -- attribute of connection object

> If this member is set to a value other than `None` it should point to a function accepting a single argument (the connection object). This will make all connection object methods work asynchronously, with the callback routine being called upon completion.

Note: for reasons beyond my understanding the callback routine is currently never called. You are advised against using asynchronous calls for the time being.

Open *(timeout)* -- Method on connection object

> Open an outgoing connection, waiting at most *timeout* seconds for the connection to be established.

Listen *(timeout)* -- Method on connection object

> Wait for an incoming connection. Stop waiting after *timeout* seconds. This call is only meaningful to some tools.

accept *(yesno)* -- Method on connection object

> Accept (when *yesno* is non-zero) or reject an incoming call after *Listen* returned.

Close *(timeout, now)* -- Method on connection object

> Close a connection. When *now* is zero, the close is orderly (i.e. outstanding output is flushed, etc.) with a timeout of *timeout* seconds. When *now* is non-zero the close is immediate, discarding output.

Read *(len, chan, timeout)* -- Method on connection object

> Read *len* bytes, or until *timeout* seconds have passed, from the channel *chan* (which is one of *cmData*, *cmCntl* or *cmAttn*). Return a 2-tuple: the data read and the end-of-message flag.

Write *(buf, chan, timeout, eom)* -- Method on connection object

> Write *buf* to channel *chan*, aborting after *timeout* seconds. When *eom* has the value *cmFlagsEOM* an end-of-message indicator will be written after the data (if this concept has a meaning for this communication tool). The method returns the number of bytes written.

Status *()* -- Method on connection object

> Return connection status as the 2-tuple `(sizes, flags)`. *sizes* is a 6-tuple giving the actual buffer sizes used (see *CMNew*), *flags* is a set of bits describing the state of the connection.

GetConfig *()* -- Method on connection object

> Return the configuration string of the communication tool. These configuration strings are tool-dependent, but usually easily parsed and modified.

SetConfig *(str)* -- Method on connection object

> Set the configuration string for the tool. The strings are parsed left-to-right, with later values taking precedence. This means individual configuration parameters can be modified by simply appending something like `'baud 4800'` to the end of the string returned by *GetConfig* and passing that to this method. The method returns the number of characters actually parsed by the tool before it encountered an error (or completed successfully).

Choose *()* -- Method on connection object

Present the user with a dialog to choose a communication tool and configure it. If there is an outstanding connection some choices (like selecting a different tool) may cause the connection to be aborted. The return value (one of the *choose** constants) will indicate this.

Idle *()* -- Method on connection object

Give the tool a chance to use the processor. You should call this method regularly.

Abort *()* -- Method on connection object

Abort an outstanding asynchronous *Open* or *Listen*.

Reset *()* -- Method on connection object

Reset a connection. Exact meaning depends on the tool.

Break *(length)* -- Method on connection object

Send a break. Whether this means anything, what it means and interpretation of the *length* parameter depend on the tool in use.

Boolean Operations -- Python library reference

Next: Comparisons Prev: Truth Value Testing Up: Types Top: Top

2.1.2. Boolean Operations

These are the Boolean operations, ordered by ascending priority:

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` --- (2)

Notes:

(1)

These only evaluate their second argument if needed for their outcome.

(2)

`` `not' `` has a lower priority than non-Boolean operators, so e.g. `not a == b` is interpreted as `not(a == b)`, and `a == not b` is a syntax error.

Next: Comparisons Prev: Truth Value Testing Up: Types Top: Top

Configuration Objects -- Python library reference

16.1.1. Configuration Objects

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

getqueuesize *()* -- Method on audio configuration object

Return the queue size.

setqueuesize *(size)* -- Method on audio configuration object

Set the queue size.

getwidth *()* -- Method on audio configuration object

Get the sample width.

setwidth *(width)* -- Method on audio configuration object

Set the sample width.

getchannels *()* -- Method on audio configuration object

Get the channel count.

setchannels *(nchannels)* -- Method on audio configuration object

Set the channel count.

getsampfmt *()* -- Method on audio configuration object

Get the sample format.

setsampfmt *(sampfmt)* -- Method on audio configuration object

Set the sample format.

getfloatmax *()* -- Method on audio configuration object

Get the maximum value for floating sample formats.

setfloatmax *(floatmax)* -- Method on audio configuration object

Set the maximum value for floating sample formats.

Window Objects -- Python library reference

Next: Drawing Objects Prev: STDWIN Functions Up: stdwin Top: Top

15.1.2. Window Objects

Window objects are created by `stdwin.open()`. They are closed by their `close()` method or when they are garbage-collected. Window objects have the following methods:

begindrawing *()* -- Method on window

> Return a drawing object, whose methods (described below) allow drawing in the window.

change *(rect)* -- Method on window

> Invalidate the given rectangle; this may cause a draw event.

gettitle *()* -- Method on window

> Returns the window's title string.

getdocsize *()* -- Method on window

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

getorigin *()* -- Method on window

> Return a pair of integers giving the origin of the window with respect to the document.

gettitle *()* -- Method on window

> Return the window's title string.

getwinsize *()* -- Method on window

> Return a pair of integers giving the size of the window.

getwinpos *()* -- Method on window

> 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)* -- Method on window

> 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)* -- Method on window

> Scroll the given rectangle by the vector given by the point.

setdocsize *(point)* -- Method on window

> Set the size of the drawing document.

setorigin *(point)* -- Method on window

> Move the origin of the window (its upper left corner) to the given point in the document.

setselection *(i, str)* -- Method on window

> Attempt to set X11 selection number *i* to the string *str*. (See stdwin method `getselection()` for the meaning of *i*.) Return true if it succeeds. If succeeds, the window ``owns'' the selection until (a) another application takes ownership of the selection; or (b) the window is deleted; or (c) the application clears ownership by calling `stdwin.resetselection(i)`. When another application takes ownership of the selection, a `WE_LOST_SEL` event is received for no particular window and

with the selection number as detail. Ignored on the Macintosh.

settimer *(dsecs)* -- Method on window

Schedule a timer event for the window in *`dsecs/10`* seconds.

settitle *(title)* -- Method on window

Set the window's title string.

setwincursor *(name)* -- Method on window

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)* -- Method on window

Set the the position of the window's upper left corner (relative to the upper left corner of the screen).

setwinsize *(width, height)* -- Method on window

Set the window's size.

show *(rect)* -- Method on window

Try to ensure that the given rectangle of the document is visible in the window.

textcreate *(rect)* -- Method on window

Create a text-edit object in the document at the given rectangle. Methods of text-edit objects are described below.

setactive *()* -- Method on window

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 *()* -- Method on window

Discard the window object. It should not be used again.

FInfo objects -- Python library reference

Prev: alias objects Up: macfs Top: Top

14.6.3. FInfo objects

See Inside Mac for a complete description of what the various fields mean.

Creator -- attribute of FInfo object

> The 4-char creator code of the file.

Type -- attribute of FInfo object

> The 4-char type code of the file.

Flags -- attribute of FInfo object

> The finder flags for the file as 16-bit integer. The bit values in *Flags* are defined in standard module *MACFS*.

Location -- attribute of FInfo object

> A Point giving the position of the file's icon in its folder.

Fldr -- attribute of FInfo object

> The folder the file is in (as an integer).

Prev: alias objects Up: macfs Top: Top

The Python Debugger -- Python library reference

Next: The Python Profiler Prev: UNIX Specific Services Up: Top Top: Top

9. The Python Debugger

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

The debugger is extensible --- it is actually defined as a class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the (also undocumented) modules `bdb` and `cmd`. A primitive windowing version of the debugger also exists --- this is module `wdb`, which requires STDWIN (see the chapter on STDWIN specific modules). The debugger's prompt is ``(Pdb) ''. Typical usage to run a program under control of the debugger is:

```
>>> import pdb

>>> import mymodule

>>> pdb.run('mymodule.test()')

> <string>(0)?()

(Pdb) continue

> <string>(1)?()

(Pdb) continue

NameError: 'spam'

> <string>(1)?()

(Pdb)
```

Typical usage to inspect a crashed program is:

```
>>> import pdb

>>> import mymodule
```

```
>>> mymodule.test()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print spam
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print spam
(Pdb)
```

The module defines the following functions; each enters the debugger in a slightly different way:

run *(statement[, globals[, locals]])* -- function of module pdb

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

runeval *(expression[, globals[, locals]])* -- function of module pdb

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

runcall *(function[, argument, ...])* -- function of module pdb

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

set_trace *()* -- function of module pdb

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

post_mortem *(traceback)* -- function of module pdb

Enter post-mortem debugging of the given *traceback* object.

pm *()* -- function of module pdb

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

Menu

notes -- Python library reference

11.12.1. notes

There is an alternative, more powerful interface to the coder and decoder, see the source for details.

If you code or decode textfiles on non-Macintosh platforms they will still use the macintosh newline convention (carriage-return as end of line).

As of this writing, *hexbin* appears to not work in all cases.

al -- Python library reference

16.1. Built-in Module al

This module provides access to the audio facilities of the SGI Indy and Indigo workstations. See section 3A of the IRIX man pages for details. You'll need to read those man pages to understand what these functions do! Some of the functions are not available in IRIX releases before 4.0.5. Again, see the manual to check whether a specific function is available on your platform.

All functions and methods defined in this module are equivalent to the C functions with `` `AL' `` prefixed to their name.

Symbolic constants from the C header file <audio.h> are defined in the standard module `AL`, see below.

Warning: the current version of the audio library may dump core when bad argument values are passed rather than returning an error status. Unfortunately, since the precise circumstances under which this may happen are undocumented and hard to check, the Python interface can provide no protection against this kind of problems. (One example is specifying an excessive queue size --- there is no documented upper limit.)

The module defines the following functions:

openport *(name, direction[, config])* -- function of module al

> The name and direction arguments are strings. The optional config argument is a configuration object as returned by `al.newconfig()`. The return value is an port object; methods of port objects are described below.

newconfig *()* -- function of module al

> The return value is a new configuration object; methods of configuration objects are described below.

queryparams *(device)* -- function of module al

> The device argument is an integer. The return value is a list of integers containing the data returned by ALqueryparams().

getparams *(device, list)* -- function of module al

> 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)* -- function of module al

> The device argument is an integer. The list argument is a list such as returned by `al.queryparams`.

Menu

Configuration Objects

Port Objects

NNTP Objects -- Python library reference

11.6.1. NNTP Objects

NNTP instances have the following methods. The *response* that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

getwelcome *()* -- Method on NNTP object

> Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

set_debuglevel *(level)* -- Method on NNTP object

> Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

newgroups *(date, time)* -- Method on NNTP object

> Send a `NEWGROUPS' command. The *date* argument should be a string of the form `"yymmdd"` indicating the date, and *time* should be a string of the form `"hhmmss"` indicating the time. Return a pair `(response, groups)` where *groups* is a list of group names that are new since the given date and time.

newnews *(group, date, time)* -- Method on NNTP object

> Send a `NEWNEWS' command. Here, *group* is a group name or `"*"`, and *date* and *time* have the same meaning as for `newgroups()`. Return a pair `(response, articles)` where *articles* is a list of article ids.

list *()* -- Method on NNTP object

> Send a `LIST' command. Return a pair `(response, list)` where *list* is a list of tuples. Each tuple has the form `(group, last, first, flag)`, where *group* is a group name, *last* and *first* are the last and first article numbers (as strings), and *flag* is `'y'` if posting is allowed, `'n'` if not, and `'m'` if the newsgroup is moderated. (Note the ordering: *last*, *first*.)

group *(name)* -- Method on NNTP object

> Send a `GROUP' command, where *name* is the group name. Return a tuple `(response, count, first, last, name)` where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name. The numbers are returned as strings.

help *()* -- Method on NNTP object

> Send a `HELP' command. Return a pair `(response, list)` where *list* is a list of help strings.

stat *(id)* -- Method on NNTP object

> Send a `STAT' command, where *id* is the message id (enclosed in `<' and `>') or an article number (as a string). Return a triple `(varresponse, number, id)` where *number* is the article number (as a string) and *id* is the article id (enclosed in `<' and `>').

next *()* -- Method on NNTP object

Send a `NEXT' command. Return as for `stat()`.

last *()* -- Method on NNTP object

Send a `LAST' command. Return as for `stat()`.

head *(id)* -- Method on NNTP object

Send a `HEAD' command, where *id* has the same meaning as for `stat()`. Return a pair `(response, list)` where *list* is a list of the article's headers (an uninterpreted list of lines, without trailing newlines).

body *(id)* -- Method on NNTP object

Send a `BODY' command, where *id* has the same meaning as for `stat()`. Return a pair `(response, list)` where *list* is a list of the article's body text (an uninterpreted list of lines, without trailing newlines).

article *(id)* -- Method on NNTP object

Send a `ARTICLE' command, where *id* has the same meaning as for `stat()`. Return a pair `(response, list)` where *list* is a list of the article's header and body text (an uninterpreted list of lines, without trailing newlines).

slave *()* -- Method on NNTP object

Send a `SLAVE' command. Return the server's *response*.

xhdr *(header, string)* -- Method on NNTP object

Send an `XHDR' command. This command is not defined in the RFC but is a common extension. The *header* argument is a header keyword, e.g. `"subject"`. The *string* argument should have the form `"first-last"` where *first* and *last* are the first and last article numbers to search. Return a pair `(response, list)`, where *list* is a list of pairs `(id, text)`, where *id* is an article id (as a string) and *text* is the text of the requested header for that article.

post *(file)* -- Method on NNTP object

Post an article using the `POST' command. The *file* argument is an open file object which is read until EOF using its `readline()` method. It should be a well-formed news article, including the required headers. The `post()` method automatically escapes lines beginning with `.'.

ihave *(id, file)* -- Method on NNTP object

Send an `IHAVE' command. If the response is not an error, treat *file* exactly as for the `post()` method.

quit *()* -- Method on NNTP object

Send a `QUIT' command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

CGI Example -- Python library reference

11.1.1. Example

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

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

```
<TITLE>Test Form Input</TITLE>


<H1>Test Form Input</H1>


<FORM METHOD="POST" ACTION="/cgi-bin/test.py">


<INPUT NAME=Name> (Name)<br>


<INPUT NAME=Address> (Address)<br>


<INPUT TYPE=SUBMIT>


</FORM>
```

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

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

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



import cgi



print "Content-type: text/html"
```

```
print                                   # End of headers!

print "<TITLE>Test Form Output</TITLE>"

print "<H1>Test Form Output</H1>"



form = cgi.SvFormContentDict()          # Load the form



name = addr = None                      # Default: no name and address



# Extract name and address from the form, if given



if form.has_key('Name'):

        name = form['Name']

if form.has_key('Address'):

        addr = form['Address']



# Print an unnumbered list of the name and address, if present



print "<UL>"

if name is not None:
```

```
        print "<LI>Name:", cgi.escape(name)

if addr is not None:

        print "<LI>Address:", cgi.escape(addr)

print "</UL>"
```

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

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

Test Form Output

- Name: *the name you entered*
- Address: *the address you entered*

If you didn't enter a name or address, the corresponding line will be missing (since the browser doesn't send empty form fields to the server).

Prev: cgi Up: cgi Top: Top

Other Built-in Types -- Python library reference

2.1.7. Other Built-in Types

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

Menu

shelve -- Python library reference

3.5. Standard Module shelve

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

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

```
import shelve

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

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

d.close()         # close it
```

Restrictions:

> The choice of which database package will be used (e.g. dbm or gdbm) depends on which interface is available. Therefore it isn't safe to open the database directly using dbm. The database is also (unfortunately) subject to the limitations of dbm, if it is used --- this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.

Dependent on the implementation, closing a persistent dictionary may or may not be necessary to flush changes to disk.

The `shelve` module does not support *concurrent* read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. UNIX file locking can be used to solve this, but this differs across UNIX versions and requires knowledge about the database implementation used.

Next: copy Prev: pickle Up: Python Services Top: Top

Bit-string Operations -- Python library reference

2.1.4.1. Bit-string Operations on Integer Types

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

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

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

Operation

Result --- *Notes*

`x | y`

bitwise or of *x* and *y*`x ^ y`

bitwise exclusive or of *x* and *y*`x & y`

bitwise and of *x* and *y*`x << n`

x shifted left by *n* bits --- (1), (2)

`x >> n`

x shifted right by *n* bits --- (1), (3) @hline@hline

`~x`

the bits of *x* inverted

Notes:

(1)

Negative shift counts are illegal.

(2)

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

(3)

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

Example -- Python library reference

3.9.2. Example

A simple example:

```
>>> import parser
```

>>> ast = parser.expr('a + 5')

>>> code = parser.compileast(ast)

>>> a = 5

>>> eval(code)

10

urlparse -- Python library reference

11.7. Standard Module urlparse

This module defines a standard interface to break URL strings up in components (addessing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a ``relative URL'' to an absolute URL given a ``base URL''.

The module has been designed to match the current Internet draft on Relative Uniform Resource Locators (and discovered a bug in an earlier draft!).

It defines the following functions:

urlparse *(urlstring[, default_scheme[, allow_fragments]])* -- function of module urlparse

> Parse a URL into 6 components, returning a 6-tuple: (addressing scheme, network location, path, parameters, query, fragment identifier). This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (e.g. the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the tuple items, except for a leading slash in the *path* component, which is retained if present.

Example:

```
urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
```

yields the tuple

```
('http', 'www.cwi.nl:80', '/%7Eguido/Python.html', '', '', '')
```

If the *default_scheme* argument is specified, it gives the default addressing scheme, to be used only if the URL string does not specify one. The default value for this argument is the empty string.

If the *allow_fragments* argument is zero, fragment identifiers are not allowed, even if the URL's addressing scheme normally does support them. The default value for this argument is `1`.

urlunparse *(tuple)* -- function of module urlparse

> Construct a URL string from a tuple as returned by `urlparse`. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had redundant delimiters, e.g. a ? with an empty query (the draft states that these are equivalent).

urljoin *(base, url[, allow_fragments])* -- function of module urlparse

> Construct a full (``absolute'') URL by combining a ``base URL'' (*base*) with a ``relative URL'' (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL.

Example:

```
urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
```

yields the string

`'http://www.cwi.nl/%7Eguido/FAQ.html'`

The *allow_fragments* argument has the same meaning as for `urlparse`.

Next: htmllib Prev: nntplib Up: Internet and WWW Top: Top

Mapping Types -- Python library reference

2.1.6. Mapping Types

A mapping object maps values of one type (the key type) to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the 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:,varvalue` 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.

Sequence Types -- Python library reference

2.1.5. Sequence Types

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

Strings literals are written in single or double quotes: `'xyzzy'`, `"frobozz"`. See Chapter 2 of the Python Reference Manual for more about string literals. Lists are constructed with square brackets, separating items with commas: `[a, b, c]`. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, e.g., `a, b, c` or `()`. A single item tuple must have a trailing comma, e.g., `(d,)`. Sequence types support the following operations. The \``in`' and \``not,in`' operations have the same priorities as the comparison operations. The \``+`' and \``*`' operations have the same priority as the corresponding numeric operations.(1)

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

Operation

Result --- *Notes*

`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)`len(s)`

length of *s*

`min(s)`

smallest item of *s*

`max(s)`

largest item of *s*

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.

Menu

---------- Footnotes ----------

(1) They must have since the parser can't tell the type of the operands.

Exceptions -- Python library reference

Next: Built-in Functions Prev: Types Up: Built-in Objects Top: Top

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` -- built-in exception

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

`EOFError` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

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

`KeyboardInterrupt` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

> 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` -- built-in exception

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.

Next: Built-in Functions Prev: Types Up: Built-in Objects Top: Top

Next: ctb Prev: mac Up: Macintosh Specific Services Top: Top

14.2. Standard Module macpath

This module provides a subset of the pathname manipulation functions available from the optional standard module `posixpath`. It is best accessed through the more portable standard module `os`, as `os.path`.

The following functions are available in this module: `normcase`, `isabs`, `join`, `split`, `isdir`, `isfile`, `exists`.

Next: ctb Prev: mac Up: Macintosh Specific Services Top: Top

Next: SGI IRIX Specific Services Prev: Macintosh Specific Services Up: Top Top: Top

15. Standard Windowing Interface

The modules in this chapter are available only on those systems where the STDWIN library is available. STDWIN runs on UNIX under X11 and on the Macintosh. See CWI report CS-R8817.

Warning: Using STDWIN is not recommended for new applications. It has never been ported to Microsoft Windows or Windows NT, and for X11 or the Macintosh it lacks important functionality --- in particular, it has no tools for the construction of dialogs. For most platforms, alternative, native solutions exist (though none are currently documented in this manual): Tkinter for UNIX under X11, native Xt with Motif or Athena widgets for UNIX under X11, Win32 for Windows and Windows NT, and a collection of native toolkit interfaces for the Macintosh.

Menu

Next: SGI IRIX Specific Services Prev: Macintosh Specific Services Up: Top Top: Top

posixpath -- Python library reference

8.2. Standard Module posixpath

This module implements some useful functions on POSIX pathnames.

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

basename *(p)* -- function of module posixpath

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

commonprefix *(list)* -- function of module posixpath

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

exists *(p)* -- function of module posixpath

> Return true if *p* refers to an existing path.

expanduser *(p)* -- function of module posixpath

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

expandvars *(p)* -- function of module posixpath

> Return the argument with environment variables expanded. Substrings of the form \``$name`' or \``${name}`' are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

isabs *(p)* -- function of module posixpath

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

isfile *(p)* -- function of module posixpath

> 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)* -- function of module posixpath

> 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)* -- function of module posixpath

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

ismount *(p)* -- function of module posixpath

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

join *(p, q)* -- function of module posixpath

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)* -- function of module posixpath

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)* -- function of module posixpath

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)* -- function of module posixpath

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)* -- function of module posixpath

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)* -- function of module posixpath

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)`, so including `` `.' `` and `` `..' ``). 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.)

string -- Python library reference

4.1. Standard Module string

This module defines some constants useful for checking character classes and some useful string functions. See the modules `regex` and `regsub` for string functions based on regular expressions.

The constants defined in this module are are:

digits -- data of module string

> The string `'0123456789'`.

hexdigits -- data of module string

> The string `'0123456789abcdefABCDEF'`.

letters -- data of module string

> The concatenation of the strings `lowercase` and `uppercase` described below.

lowercase -- data of module string

> 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 -- data of module string

> The string `'01234567'`.

uppercase -- data of module string

> 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 -- data of module string

> A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab. Do not change its definition --- the effect on the routines `strip` and `split` is undefined.

The functions defined in this module are:

atof *(s)* -- function of module string

> 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[, base])* -- function of module string

> Convert string *s* to an integer in the given *base*. The string must consist of one or more digits, optionally preceded by a sign (\``+`' or \``-`'). The *base* defaults to 10. If it is 0, a default base is chosen depending on the leading characters of the string (after stripping the sign): \``0x`' or \``0X`' means 16, \``0`' means 8, anything else means 10. If *base* is 16, a leading \``0x`' or \``0X`' is always accepted. (Note: for a more flexible interpretation of numeric literals, use the built-in function `eval()`.)

atol *(s[, base])* -- function of module string

> Convert string *s* to a long integer in the given *base*. The string must consist of one or more digits, optionally preceded by a sign (\``+`' or \``-`'). The *base* argument has the same meaning as for

`atoi()`. A trailing `` `l' `` or `` `L' `` is not allowed.

expandtabs *(s, tabsize)* -- function of module string

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[, start])* -- function of module string

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

rfind *(s, sub[, start])* -- function of module string

Like `find` but find the highest index.

index *(s, sub[, start])* -- function of module string

Like `find` but raise `ValueError` when the substring is not found.

rindex *(s, sub[, start])* -- function of module string

Like `rfind` but raise `ValueError` when the substring is not found.

count *(s, sub[, start])* -- function of module string

Return the number of (non-overlapping) occurrences of substring *sub* in string *s* with index at least *start*. If *start* is omitted, it defaults to `0`. If *start* is negative, `len(s)` is added.

lower *(s)* -- function of module string

Convert letters to lower case.

split *(s)* -- function of module string

Return a list of the whitespace-delimited words of the string *s*.

splitfields *(s, sep)* -- function of module string

Return 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)* -- function of module string

Concatenate a list or tuple of words with intervening spaces.

joinfields *(words, sep)* -- function of module string

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)* -- function of module string

Remove leading and trailing whitespace from the string *s*.

swapcase *(s)* -- function of module string

Convert lower case letters to upper case and vice versa.

translate *(s, table)* -- function of module string

Translate the characters from *s* using *table*, which must be a 256-character string giving the translation for each character value, indexed by its ordinal.

upper *(s)* -- function of module string

> Convert letters to upper case.

ljust *(s, width)* -- function of module string

rjust *(s, width)* -- function of module string

center *(s, width)* -- function of module string

> 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)* -- function of module string

> Pad a numeric string on the left with zero digits until the given width is reached. Strings starting with a sign are handled correctly.

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

Next: regex Prev: String Services Up: String Services Top: Top

pwd -- Python library reference

8.3. 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)* -- function of module pwd

> Return the password database entry for the given numeric user ID.

getpwnam *(name)* -- function of module pwd

> Return the password database entry for the given user name.

getpwall *()* -- function of module pwd

> Return a list of all available password database entries, in arbitrary order.

imageop -- Python library reference

12.2. Built-in Module imageop

The `imageop` module contains some useful operations on images. It operates on images consisting of 8 or 32 bit 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 -- exception of module imageop

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)* -- function of module imageop

Return the selected part of *image*, which should by *width* by *height* in size and consist of pixels of *psize* bytes. *x0*, *y0*, *x1* and *y1* are like the `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)* -- function of module imageop

Return *image* scaled to size *newwidth* by *newheight*. No interpolation is done, scaling is done by simple-minded pixel duplication or removal. Therefore, computer-generated images or dithered images will not look nice after scaling.

tovideo *(image, psize, width, height)* -- function of module imageop

Run a vertical low-pass filter over an image. It does so by computing each destination pixel as the average of two vertically-aligned source pixels. The main use of this routine is to forestall excessive flicker if the image is displayed on a video device that uses interlacing, hence the name.

grey2mono *(image, width, height, threshold)* -- function of module imageop

Convert 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)* -- function of module imageop

Convert an 8-bit greyscale image to a 1-bit monochrome image using a (simple-minded) dithering algorithm.

mono2grey *(image, width, height, p0, p1)* -- function of module imageop

Convert a 1-bit monochrome image to an 8 bit greyscale or color image. All pixels that are zero-valued on input get value *p0* on output and all one-value input pixels get value *p1* on output. To convert a monochrome black-and-white image to greyscale pass the values `0` and `255` respectively.

grey2grey4 *(image, width, height)* -- function of module imageop

Convert an 8-bit greyscale image to a 4-bit greyscale image without dithering.

grey2grey2 *(image, width, height)* -- function of module imageop

Convert an 8-bit greyscale image to a 2-bit greyscale image without dithering.

dither2grey2 *(image, width, height)* -- function of module imageop

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)* -- function of module imageop

Convert a 4-bit greyscale image to an 8-bit greyscale image.

grey22grey *(image, width, height)* -- function of module imageop

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

regex -- Python library reference

4.2. 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, with one exception. There is a way to change the syntax to match that of several well-known UNIX utilities. The exception is that Emacs' `` `\s' `` pattern is not supported, since the original implementation references the Emacs syntax tables.

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)* -- function of module regex

> 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)* -- function of module regex

> 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])* -- function of module regex

> 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 argument *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`. This can be used to implement case-insensitive matching; see the `casefold` data item below.

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)* -- function of module regex

> 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])* -- function of module regex

> This is like `compile`, but supports symbolic group names: if a parenthesis-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')`. Group names may contain alphanumeric characters and `'_'` only.

error -- exception of module regex

> 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 -- data of module regex

> 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])* -- Method on regex

> 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])* -- Method on regex

> 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, ...)* -- Method on regex

> 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 -- attribute of regex

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 -- attribute of regex

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 -- attribute of regex

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 -- attribute of regex

The regular expression pattern as passed to `compile` or `symcomp`.

realpat -- attribute of regex

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

groupindex -- attribute of regex

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

copy -- Python library reference

3.6. Standard Module copy

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

Interface summary:

```
import copy
```

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

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

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

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

> A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.

A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

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

> Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.

Because deep copy copies *everything* it may copy too much, e.g. administrative data structures that should be shared even between copies.

Python's `deepcopy()` operation avoids these problems by:

> keeping a table of objects already copied during the current copying pass; and

letting user-defined classes override the copying operation or the set of components copied.

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

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

How It Works -- Python library reference

9.2. How It Works

Some changes were made to the interpreter:

sys.settrace(func) sets the global trace function

there can also a local trace function (see later)

Trace functions have three arguments: (*frame*, *event*, *arg*)

frame

is the current stack frame

event

is a string: `'call'`, `'line'`, `'return'` or `'exception'`

arg

is dependent on the event type

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

The events have the following meaning:

`'call'`

A function is called (or some other code block entered). The global trace function is called; arg is the argument list to the function; the return value specifies the local trace function.

`'line'`

The interpreter is about to execute a new line of code (sometimes multiple line events on one line exist). The local trace function is called; arg in None; the return value specifies the new local trace function.

`'return'`

A function (or other code block) is about to return. The local trace function is called; arg is the value that will be returned. The trace function's return value is ignored.

`'exception'`

An exception has occurred. The local trace function is called; arg is a triple (exception, value, traceback); the return value specifies the new local trace function

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

Stack frame objects have the following read-only attributes:

f_code

the code object being executed

f_lineno

the current line number (`-1` for `'call'` events)

f_back

the stack frame of the caller, or None

f_locals

dictionary containing local name bindings

f_globals

dictionary containing global name bindings

Code objects have the following read-only attributes:

co_code

the code string

co_names

the list of names used by the code

co_consts

the list of (literal) constants used by the code

co_filename

the filename from which the code was compiled

types -- Python library reference

Next: traceback Prev: sys Up: Python Services Top: Top

3.2. Standard Module types

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

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

```
from types import *

def delete(list, item):

    if type(item) is IntType:

        del list[item]

    else:

        list.remove(item)
```

The module defines the following names:

NoneType -- data of module types

The type of `None`.

TypeType -- data of module types

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

IntType -- data of module types

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

LongType -- data of module types

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

FloatType -- data of module types

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

StringType -- data of module types

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

TupleType -- data of module types

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

ListType -- data of module types

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

DictType -- data of module types

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

DictionaryType -- data of module types

An alternative name for `DictType`.

FunctionType -- data of module types

The type of user-defined functions and lambdas.

LambdaType -- data of module types

An alternative name for `FunctionType`.

CodeType -- data of module types

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

ClassType -- data of module types

The type of user-defined classes.

InstanceType -- data of module types

The type of instances of user-defined classes.

MethodType -- data of module types

The type of methods of user-defined class instances.

UnboundMethodType -- data of module types

An alternative name for `MethodType`.

BuiltinFunctionType -- data of module types

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

BuiltinMethodType -- data of module types

An alternative name for `BuiltinFunction`.

ModuleType -- data of module types

The type of modules.

FileType -- data of module types

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

XRangeType -- data of module types

The type of range objects returned by `xrange()`.

TracebackType -- data of module types

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

FrameType -- data of module types

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

TCP Stream Objects -- Python library reference

14.9.1. TCP Stream Objects

asr -- attribute of TCP stream

> When set to a value different than `None` this should point to a function with two integer parameters: an event code and a detail. This function will be called upon network-generated events such as urgent data arrival. In addition, it is called with eventcode `MACTCP.PassiveOpenDone` when a `PassiveOpen` completes. This is a Python addition to the MacTCP semantics. It is safe to do further calls from the `asr`.

PassiveOpen *(port)* -- Method on TCP stream

> Wait for an incoming connection on TCP port *port* (zero makes the system pick a free port). The call returns immediately, and you should use *wait* to wait for completion. You should not issue any method calls other than `wait`, `isdone` or `GetSockName` before the call completes.

wait *()* -- Method on TCP stream

> Wait for `PassiveOpen` to complete.

isdone *()* -- Method on TCP stream

> Return 1 if a `PassiveOpen` has completed.

GetSockName *()* -- Method on TCP stream

> Return the TCP address of this side of a connection as a 2-tuple `(host, port)`, both integers.

ActiveOpen *(lport, host, rport)* -- Method on TCP stream

> Open an outgoing connection to TCP address `(host, rport)`. Use local port *lport* (zero makes the system pick a free port). This call blocks until the connection has been established.

Send *(buf, push, urgent)* -- Method on TCP stream

> Send data *buf* over the connection. *Push* and *urgent* are flags as specified by the TCP standard.

Rcv *(timeout)* -- Method on TCP stream

> Receive data. The call returns when *timeout* seconds have passed or when (according to the MacTCP documentation) ``a reasonable amount of data has been received''. The return value is a 3-tuple `(data, urgent, mark)`. If urgent data is outstanding `Rcv` will always return that before looking at any normal data. The first call returning urgent data will have the *urgent* flag set, the last will have the *mark* flag set.

Close *()* -- Method on TCP stream

> Tell MacTCP that no more data will be transmitted on this connection. The call returns when all data has been acknowledged by the receiving side.

Abort *()* -- Method on TCP stream

> Forcibly close both sides of a connection, ignoring outstanding data.

Status *()* -- Method on TCP stream

> Return a TCP status object for this stream giving the current status (see below).

String Services -- Python library reference

4. String Services

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

string

--- Common string operations.

regex

--- Regular expression search and match operations.

regsub

--- Substitution and splitting operations that use regular expressions.

struct

--- Interpret strings as packed binary data.

Menu

audioop -- Python library reference

12.1. Built-in Module audioop

The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16 or 32 bits wide, stored in Python strings. This is the same format as used by the `al` and `sunaudiodev` modules. All scalar items are integers, unless specified otherwise.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

error -- exception of module audioop

> This exception is raised on all errors, such as unknown number of bytes per sample, etc.

add *(fragment1, fragment2, width)* -- function of module audioop

> Return a fragment which is the addition of the two samples passed as parameters. *width* is the sample width in bytes, either `1`, `2` or `4`. Both fragments should have the same length.

adpcm2lin *(adpcmfragment, width, state)* -- function of module audioop

> Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm` for details on ADPCM coding. Return a tuple `(sample, newstate)` where the sample has the width specified in *width*.

adpcm32lin *(adpcmfragment, width, state)* -- function of module audioop

> Decode an alternative 3-bit ADPCM code. See `lin2adpcm3` for details.

avg *(fragment, width)* -- function of module audioop

> Return the average over all samples in the fragment.

avgpp *(fragment, width)* -- function of module audioop

> Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

bias *(fragment, width, bias)* -- function of module audioop

> Return a fragment that is the original fragment with a bias added to each sample.

cross *(fragment, width)* -- function of module audioop

> Return the number of zero crossings in the fragment passed as an argument.

findfactor *(fragment, reference)* -- function of module audioop

> Return a factor *F* such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

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

findfit *(fragment, reference)* -- function of module audioop

> This routine (which only accepts 2-byte sample fragments)

Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using `findfactor` to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple `(offset, factor)` where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per `findfactor`.

findmax *(fragment, length)* -- function of module audioop

Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return *i* for which `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

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

getsample *(fragment, width, index)* -- function of module audioop

Return the value of sample *index* from the fragment.

lin2lin *(fragment, width, newwidth)* -- function of module audioop

Convert samples between 1-, 2- and 4-byte formats.

lin2adpcm *(fragment, width, state)* -- function of module audioop

Convert 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 algorithm has been selected for use by the IMA, so it may well become a standard.

`State` is a tuple containing the state of the coder. The coder returns a tuple `(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)* -- function of module audioop

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)* -- function of module audioop

Convert samples in the audio fragment to U-LAW encoding and return this as a Python string. U-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

minmax *(fragment, width)* -- function of module audioop

Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

max *(fragment, width)* -- function of module audioop

Return the maximum of the *absolute value* of all samples in a fragment.

maxpp *(fragment, width)* -- function of module audioop

Return the maximum peak-peak value in the sound fragment.

mul *(fragment, width, factor)* -- function of module audioop

Return a fragment that has all samples in the original framgent multiplied by the floating-point value *factor*. Overflow is silently ignored.

reverse *(fragment, width)* -- function of module audioop

Reverse the samples in a fragment and returns the modified fragment.

rms *(fragment, width)* -- function of module audioop

Return the root-mean-square of the fragment, i.e. the square root of the quotient of the sum of all squared sample value, divided by the sumber of samples. This is a measure of the power in an audio signal.

tomono *(fragment, width, lfactor, rfactor)* -- function of module audioop

Convert 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)* -- function of module audioop

Generate 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*.

ulaw2lin *(fragment, width)* -- function of module audioop

Convert 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 to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):

    pos = audioop.findmax(outputdata, 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)
```

Examples -- Python library reference

Prev: imp Up: imp Top: Top

3.8.1. Examples

The following function emulates the default import statement:

```
import imp

import sys



def __import__(name, globals=None, locals=None, fromlist=None):

    # Fast path: see if the module has already been imported.

    if sys.modules.has_key(name):

        return sys.modules[name]



    # If any of the following calls raises an exception,

    # there's a problem we can't handle -- let the caller handle it.



    # See if it's a built-in module.

    m = imp.init_builtin(name)

    if m:

        return m
```

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

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

# See what we got.
try:
    if type == imp.C_EXTENSION:
        return imp.load_dynamic(name, pathname)
    if type == imp.PY_SOURCE:
        return imp.load_source(name, pathname, fp)
    if type == imp.PY_COMPILED:
        return imp.load_compiled(name, pathname, fp)

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

```
    finally:
        # Since we may exit via an exception, close fp explicitly.
        fp.close()
```

The Stats Class -- Python library reference

10.5.1. The Stats Class

strip_dirs *()* -- Method on Stats

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

add *(filename[, ...])* -- Method on Stats

> This method of the `Stats` class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of `profile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

sort_stats *(key[, ...])* -- Method on Stats

> This method modifies the `Stats` object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: `"time"` or `"name"`).

When more than one key is provided, then additional keys are used as secondary criteria when the there is equality in all keys selected before them. For example, sort_stats('name', 'file') will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg

--- *Meaning*

`"calls"`

call count

`"cumulative"`

cumulative time

`"file"`

file name

`"module"`

file name

`"pcalls"`

primitive call count

`"line"`

line number

`"name"`

function name

`"nfl"`

name/file/line

`"stdname"`

standard name

`"time"`

internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between `"nfl"` and `"stdname"` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `"nfl"` does a numeric compare of the line numbers. In fact, `sort_stats("nfl")` is the same as `sort_stats("name", "file", "line")`.

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

reverse_order *()* -- Method on Stats

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

print_stats *(restriction[, ...])* -- Method on Stats

This method for the `Stats` class prints out a report as described in the `profile.run()` definition.

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

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

```
print_stats(.1, "foo:")
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename \``.*foo:`'. In contrast, the command:

```
print_stats("foo:", .1)
```

would limit the list to all functions having file names \``.*foo:`', and then proceed to only print the first 10% of them.

print_callers *(restrictions[, ...])* -- Method on Stats

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

print_callees *(restrictions[, ...])* -- Method on Stats

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

ignore *()* -- Method on Stats

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

```
pstats.Stats('foofile').strip_dirs().sort_stats('cum') \
```

.print_stats().ignore()

would perform all the indicated functions, but it would not return the final reference to the `Stats` instance.(1)

---------- Footnotes ----------

(1) This was once necessary, when Python would print any unused expression result that was not `None`. The method is still defined for backward compatibility.

math -- Python library reference

5.1. Built-in Module math

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

acos *(x)* -- function of module math

asin *(x)* -- function of module math

atan *(x)* -- function of module math

atan2 *(x, y)* -- function of module math

ceil *(x)* -- function of module math

cos *(x)* -- function of module math

cosh *(x)* -- function of module math

exp *(x)* -- function of module math

fabs *(x)* -- function of module math

floor *(x)* -- function of module math

fmod *(x, y)* -- function of module math

frexp *(x)* -- function of module math

hypot *(x, y)* -- function of module math

ldexp *(x, y)* -- function of module math

log *(x)* -- function of module math

log10 *(x)* -- function of module math

modf *(x)* -- function of module math

pow *(x, y)* -- function of module math

sin *(x)* -- function of module math

sinh *(x)* -- function of module math

sqrt *(x)* -- function of module math

tan *(x)* -- function of module math

tanh *(x)* -- function of module math

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 `hypot` function, which is not standard C, is not available on all platforms.

The module also defines two mathematical constants:

pi -- data of module math

e -- data of module math

Multimedia Services -- Python library reference

12. Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here's an overview:

audioop

--- Manipulate raw audio data.

imageop

--- Manipulate raw image data.

aifc

--- Read and write audio files in AIFF or AIFC format.

jpeg

--- Read and write image files in compressed JPEG format.

rgbimg

--- Read and write image files in ``SGI RGB'' format (the module is *not* SGI specific though)!

Menu

nntplib -- Python library reference

11.6. Standard Module nntplib

This module defines the class `NNTP` which implements the client side of the NNTP protocol. It can be used to implement a news reader or poster, or automated news processors. For more information on NNTP (Network News Transfer Protocol), see Internet RFC 977.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = NNTP('news.cwi.nl')
```

>>> resp, count, first, last, name = s.group('comp.lang.python')

>>> print 'Group', name, 'has', count, 'articles, range', first, 'to', last

Group comp.lang.python has 59 articles, range 3742 to 3803

>>> resp, subs = s.xhdr('subject', first + '-' + last)

>>> for id, sub in subs[-10:]: print id, sub

...

3792 Re: Removing elements from a list while iterating...

3793 Re: Who likes Info files?

3794 Emacs and doc strings

3795 a few questions about the Mac implementation

3796 Re: executable python scripts

3797 Re: executable python scripts

3798 Re: a few questions about the Mac implementation

3799 Re: PROPOSAL: A Generic Python Object Interface for Python C Modules

3802 Re: executable python scripts

3803 Re: POSIX wait and SIGCHLD

>>> s.quit()

'205 news.cwi.nl closing connection.   Goodbye.'

>>>

}

To post an article from a file (this assumes that the article has valid headers):

```
>>> s = NNTP('news.cwi.nl')
```

>>> f = open('/tmp/article')

>>> s.post(f)

'240 Article posted successfully.'

>>> s.quit()

'205 news.cwi.nl closing connection.   Goodbye.'

>>>

The module itself defines the following items:

NNTP *(host[, port])* -- function of module nntplib

> Return a new instance of the `NNTP` class, representing a connection to the NNTP server running on host *host*, listening at port *port*. The default *port* is 119.

error_reply -- exception of module nntplib

> Exception raised when an unexpected reply is received from the server.

error_temp -- exception of module nntplib

> Exception raised when an error code in the range 400--499 is received.

error_perm -- exception of module nntplib

Exception raised when an error code in the range 500--599 is received.

error_proto -- exception of module nntplib

Exception raised when a reply is received from the server that does not begin with a digit in the range 1--5.

Menu

gl -- Python library reference

16.8. Built-in Module gl

This module provides access to the Silicon Graphics *Graphics Library*. It is available only on Silicon Graphics machines.

Warning: Some illegal calls to the GL library cause the Python interpreter to dump core. In particular, the use of most GL calls is unsafe before the first window is opened.

The module is too large to document here in its entirety, but the following should help you to get started. The parameter conventions for the C functions are translated to Python as follows:

All (short, long, unsigned) int values are represented by Python integers.

All float and double values are represented by Python floating point numbers. In most cases, Python integers are also allowed.

All arrays are represented by one-dimensional Python lists. In most cases, tuples are also allowed.

All string and character arguments are represented by Python strings, for instance, `winopen('Hi There!')` and `rotate(900, 'z')`.

All (short, long, unsigned) integer arguments or return values that are only used to specify the length of an array argument are omitted. For example, the C call

```
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)* -- function of module gl

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 *()* -- function of module gl

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 *()* -- function of module gl

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)* -- function of module gl

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)* -- function of module gl

Defines a nurbs curve. The length of ctlpoints is `len(knots) - order`.

pwlcurve *(points, type)* -- function of module gl

Defines a piecewise-linear curve. *points* is a list of points. *type* must be `N_ST`.

pick *(n)* -- function of module gl

select *(n)* -- function of module gl

The only argument to these functions specifies the desired size of the pick or select buffer.

endpick *()* -- function of module gl

endselect *()* -- function of module gl

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()
```

parser -- Python library reference

3.9. Built-in Module parser

The `parser` module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This can be better than trying to parse and modify an arbitrary Python code fragment as a string, and ensures that parsing is performed in a manner identical to the code forming the application. It's also faster.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to the Language Reference. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ``AST objects'' created by this module are the actual output from the internal parser when created by the `expr()` or `suite()` functions, described below. The AST objects created by `tuple2ast()` faithfully simulate those structures.

Each element of the tuples returned by `ast2tuple()` has a simple form. Tuples representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module `Lib/symbol.py`. Each additional element of the tuple represents a component of the production as recognized in the input string: these are always tuples which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an *if_stmt*, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` elements, including variable and function names defined by the user.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module `Lib/token.py`.

The AST objects are not actually required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple ``wrapper'' module may be created in Python if desired to hide the use of AST objects.

The `parser` module defines the following functions:

ast2tuple *(ast)* -- function of module parser

> This function accepts an AST object from the caller in *`ast`* and returns a Python tuple representing the equivelent parse tree. The resulting tuple representation can be used for inspection or the creation of a new parse tree in tuple form. This function does not fail so long as memory is available to build the tuple representation.

compileast *(ast[, filename = '<ast>'])* -- function of module parser

> The Python byte compiler can be invoked on an AST object to produce code objects which can be used as part of an `exec` statement or a call to the built-in `eval()` function. This function provides the interface to the compiler, passing the internal parse tree from *`ast`* to the parser, using the

source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an AST object.

expr *(string)* -- function of module parser

The `expr()` function parses the parameter *string* as if it were an input to `compile(string, 'eval')`. If the parse succeeds, an AST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

isexpr *(ast)* -- function of module parser

When *ast* represents an `'eval'` form, this function returns a true value (`1`), otherwise it returns false (`0`). This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compileast()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

issuite *(ast)* -- function of module parser

This function mirrors `isexpr()` in that it reports whether an AST object represents a suite of statements. It is not safe to assume that this function is equivelent to `not isexpr(ast)`, as additional syntactic fragments may be supported in the future.

suite *(string)* -- function of module parser

The `suite()` function parses the parameter *string* as if it were an input to `compile(string, 'exec')`. If the parse succeeds, an AST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

tuple2ast *(tuple)* -- function of module parser

This function accepts a parse tree represented as a tuple and builds an internal representation if possible. If it can validate that the tree conforms to the Python syntax and all nodes are valid node types in the host version of Python, an AST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a `ParserError` exception is thrown. An AST object created this way should not be assumed to compile correctly; normal exceptions thrown by compilation may still be initiated when the AST object is passed to `compileast()`. This will normally indicate problems not related to syntax (such as a `MemoryError` exception).

Menu

More String Operations -- Python library reference

Next: Mutable Sequence Types Prev: Sequence Types Up: Sequence Types Top: Top

2.1.5.1. 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.

The right argument should be a tuple with one item for each argument required by the format string; if the string requires a single argument, the right argument may also be a single non-tuple object.(1) 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 `%s` conversion takes any Python object and converts it to a string using `str()` before formatting it. The ANSI features `%p` and `%n` are not supported. Since Python strings have an explicit length, `%s` conversions don't assume that `'\0'` is the end of the string.

For safety reasons, floating point precisions are clipped to 50; `%f` conversions for numbers whose absolute value is over 1e25 are replaced by `%g` conversions.(2) All other errors raise exceptions.

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

```
>>> count = 2

>>> language = 'Python'

>>> print '%(language)s has %(count)03d quote types.' % vars()

Python has 002 quote types.

>>>
```

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

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

---------- Footnotes ----------

(1) A tuple object in this case should be a singleton.

(2) These numbers are fairly arbitrary. They are intended to avoid printing endless strings of meaningless digits without hampering correct use and without having to know the exact precision of floating point values on a particular machine.

Next: Mutable Sequence Types Prev: Sequence Types Up: Sequence Types Top: Top

rgbimg -- Python library reference

12.5. Built-in Module rgbimg

The rgbimg module allows python programs to access SGI imglib image files (also known as .rgb files). The module is far from complete, but is provided anyway since the functionality that there is is enough in some cases. Currently, colormap files are not supported.

The module defines the following variables and functions:

error -- exception of module rgbimg

> This exception is raised on all errors, such as unsupported file type, etc.

sizeofimage *(file)* -- function of module rgbimg

> 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)* -- function of module rgbimg

> 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)* -- function of module rgbimg

> 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)* -- function of module rgbimg

> 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).

Form Objects -- Python library reference

16.4.2. Form Objects

Form objects (returned by `fl.make_form()` above) have the following methods. Each method corresponds to a C function whose name is prefixed with \``fl_`'; and whose first argument is a form pointer; please refer to the official FORMS documentation for descriptions.

All the \``add_...`' functions return a Python object representing the FORMS object. Methods of FORMS objects are described below. Most kinds of FORMS object also have some methods specific to that kind; these methods are listed here.

show_form *(placement, bordertype, name)* -- Method on form object

Show the form.

hide_form *()* -- Method on form object

Hide the form.

redraw_form *()* -- Method on form object

Redraw the form.

set_form_position *(x, y)* -- Method on form object

Set the form's position.

freeze_form *()* -- Method on form object

Freeze the form.

unfreeze_form *()* -- Method on form object

Unfreeze the form.

activate_form *()* -- Method on form object

Activate the form.

deactivate_form *()* -- Method on form object

Deactivate the form.

bgn_group *()* -- Method on form object

Begin a new group of objects; return a group object.

end_group *()* -- Method on form object

End the current group of objects.

find_first *()* -- Method on form object

Find the first object in the form.

find_last *()* -- Method on form object

Find the last object in the form.

add_box *(type, x, y, w, h, name)* -- Method on form object

Add a box object to the form. No extra methods.

add_text *(type, x, y, w, h, name)* -- Method on form object

Add a text object to the form. No extra methods.

add_clock *(type, x, y, w, h, name)* -- Method on form object

Add a clock object to the form. * Method: `get_clock`.

add_button *(type, x, y, w, h, name)* -- Method on form object

Add a button object to the form. * Methods: `get_button`, `set_button`.

add_lightbutton *(type, x, y, w, h, name)* -- Method on form object

Add a lightbutton object to the form. * Methods: `get_button`, `set_button`.

add_roundbutton *(type, x, y, w, h, name)* -- Method on form object

Add a roundbutton object to the form. * Methods: `get_button`, `set_button`.

add_slider *(type, x, y, w, h, name)* -- Method on form object

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)* -- Method on form object

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)* -- Method on form object

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)* -- Method on form object

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)* -- Method on form object

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)* -- Method on form object

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)* -- Method on form object

Add a menu object to the form. * Methods: `set_menu`, `get_menu`, `addto_menu`.

add_choice *(type, x, y, w, h, name)* -- Method on form object

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)* -- Method on form object

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)* -- Method on form object

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

Audio Device Objects -- Python library reference

17.1.1. Audio Device Objects

The audio device objects are returned by `open` define the following methods (except `control` objects which only provide getinfo, setinfo and drain):

close *()* -- Method on audio device

> 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 *()* -- Method on audio device

> 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 *()* -- Method on audio device

> 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 *()* -- Method on audio device

> 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 *()* -- Method on audio device

> 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 *()* -- Method on audio device

> 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)* -- Method on audio device

> 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)* -- Method on audio device

> 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)* -- Method on audio device

> Write is passed a python string containing audio samples to be played. If there is enough buffer space free it will immediately return, otherwise it will block.

There is a companion module, `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.

Prev: sunaudiodev Up: sunaudiodev Top: Top

Macintosh Specific Services -- Python library reference

14. Macintosh Specific Services

The modules in this chapter are available on the Apple Macintosh only.

Menu

Drawing Objects -- Python library reference

15.1.3. Drawing Objects

Drawing objects are created exclusively by the window method `begindrawing()`. Only one drawing object can exist at any given time; the drawing object must be deleted to finish drawing. No drawing object may exist when `stdwin.getevent()` is called. Drawing objects have the following methods:

box *(rect)* -- Method on drawing

> Draw a box just inside a rectangle.

circle *(center, radius)* -- Method on drawing

> Draw a circle with given center point and radius.

elarc *(center, (rh, rv), (a1, a2))* -- Method on drawing

> 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)* -- Method on drawing

> Erase a rectangle.

fillcircle *(center, radius)* -- Method on drawing

> Draw a filled circle with given center point and radius.

fillelarc *(center, (rh, rv), (a1, a2))* -- Method on drawing

> Draw a filled elliptical arc; arguments as for `elarc`.

fillpoly *(points)* -- Method on drawing

> Draw a filled polygon given by a list (or tuple) of points.

invert *(rect)* -- Method on drawing

> Invert a rectangle.

line *(p1, p2)* -- Method on drawing

> Draw a line from point *p1* to *p2*.

paint *(rect)* -- Method on drawing

> Fill a rectangle.

poly *(points)* -- Method on drawing

> Draw the lines connecting the given list (or tuple) of points.

shade *(rect, percent)* -- Method on drawing

> Fill a rectangle with a shading pattern that is about *percent* percent filled.

text *(p, str)* -- Method on drawing

> Draw a string starting at point p (the point specifies the top left coordinate of the string).

xorcircle *(center, radius)* -- Method on drawing

xorelarc *(center, (rh, rv), (a1, a2))* -- Method on drawing

xorline *(p1, p2)* -- Method on drawing

xorpoly *(points)* -- Method on drawing

Draw a circle, an elliptical arc, a line or a polygon, respectively, in XOR mode.

setfgcolor *()* -- Method on drawing

setbgcolor *()* -- Method on drawing

getfgcolor *()* -- Method on drawing

getbgcolor *()* -- Method on drawing

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 *()* -- Method on drawing

baseline *()* -- Method on drawing

lineheight *()* -- Method on drawing

textbreak *()* -- Method on drawing

textwidth *()* -- Method on drawing

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)* -- Method on drawing

Draw the *bitmap* with its top left corner at *point*. If the optional *mask* argument is present, it should be either the same object as *bitmap*, to draw only those bits that are set in the bitmap, in the foreground color, or `None`, to draw all bits (ones are drawn in the foreground color, zeros in the background color). Not available on the Macintosh.

cliprect *(rect)* -- Method on drawing

Set the ``clipping region'' to a rectangle. The clipping region limits the effect of all drawing operations, until it is changed again or until the drawing object is closed. When a drawing object is created the clipping region is set to the entire window. When an object to be drawn falls partly outside the clipping region, the set of pixels drawn is the intersection of the clipping region and the set of pixels that would be drawn by the same operation in the absence of a clipping region.

noclip *()* -- Method on drawing

Reset the clipping region to the entire window.

close *()* -- Method on drawing

enddrawing *()* -- Method on drawing

Discard the drawing object. It should not be used again.

cgi -- Python library reference

11.1. Standard Module cgi

This module makes it easy to write Python scripts that run in a WWW server using the Common Gateway Interface. It was written by Michael McLay and subsequently modified by Steve Majewski and Guido van Rossum.

When a WWW server finds that a URL contains a reference to a file in a particular subdirectory (usually `/cgibin`), it runs the file as a subprocess. Information about the request such as the full URL, the originating host etc., is passed to the subprocess in the shell environment; additional input from the client may be read from standard input. Standard output from the subprocess is sent back across the network to the client as the response from the request. The CGI protocol describes what the environment variables passed to the subprocess mean and how the output should be formatted. The official reference documentation for the CGI protocol can be found on the World-Wide Web at `<URL:http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>`. The `cgi` module was based on version 1.1 of the protocol and should also work with version 1.0.

The `cgi` module defines several classes that make it easy to access the information passed to the subprocess from a Python script; in particular, it knows how to parse the input sent by an HTML ``form'' using either a POST or a GET request (these are alternatives for submitting forms in the HTTP protocol).

The formatting of the output is so trivial that no additional support is needed. All you need to do is print a minimal set of MIME headers describing the output format, followed by a blank line and your actual output. E.g. if you want to generate HTML, your script could start as follows:

```
# Header -- one or more lines:

print "Content-type: text/html"

# Blank line separating header from body:

print

# Body, in HTML format:

print "<TITLE>The Amazing SPAM Homepage!</TITLE>"

# etc...
```

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

The `cgi` module defines the following functions:

parse *()* -- function of module cgi

Read and parse the form submitted to the script and return a dictionary containing the form's fields.

This should be called at most once per script invocation, as it may consume standard input (if the form was submitted through a POST request). The keys in the resulting dictionary are the field names used in the submission; the values are *lists* of the field values (since field name may be used multiple times in a single form). `%' escapes in the values are translated to their single-character equivalent using urllib.unquote(). As a side effect, this function sets environ['QUERY_STRING'] to the raw query string, if it isn't already set.

print_environ_usage *()* -- function of module cgi

Print a piece of HTML listing the environment variables that may be set by the CGI protocol. This is mainly useful when learning about writing CGI scripts.

print_environ *()* -- function of module cgi

Print a piece of HTML text showing the entire contents of the shell environment. This is mainly useful when debugging a CGI script.

print_form *(form)* -- function of module cgi

Print a piece of HTML text showing the contents of the *form* (a dictionary, an instance of the FormContentDict class defined below, or a subclass thereof). This is mainly useful when debugging a CGI script.

escape *(string)* -- function of module cgi

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

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

FormContentDict *()* -- function of module cgi

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

SvFormContentDict *()* -- function of module cgi

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

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

The module defines the following variable:

environ -- data of module cgi

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

Menu

CGI Example

Text-edit Objects -- Python library reference

15.1.6. Text-edit Objects

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

arrow *(code)* -- Method on text-edit

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)* -- Method on text-edit

Pass a draw event to the text-edit block. The rectangle specifies the redraw area.

event *(type, window, detail)* -- Method on text-edit

Pass an event gotten from `stdwin.getevent()` to the text-edit block. Return true if the event was handled.

getfocus *()* -- Method on text-edit

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

getfocustext *()* -- Method on text-edit

Return the text in the focus.

getrect *()* -- Method on text-edit

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 *()* -- Method on text-edit

Return the entire text buffer.

move *(rect)* -- Method on text-edit

Specify a new position for the text-edit block in the document.

replace *(str)* -- Method on text-edit

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)* -- Method on text-edit

Specify the new focus. Out-of-bounds values are silently clipped.

settext *(str)* -- Method on text-edit

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

setview *(rect)* -- Method on text-edit

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 *()* -- Method on text-edit

Discard the text-edit object. It should not be used again.

Methods -- Python library reference

Next: Code Objects Prev: Functions Up: Other Built-in Types Top: Top

2.1.7.4. 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.)

Next: Code Objects Prev: Functions Up: Other Built-in Types Top: Top

macostools -- Python library reference

Next: mactcp Prev: MacOS Up: Macintosh Specific Services Top: Top

14.8. Standard module macostools

This module contains some convenience routines for file-manipulation on the Macintosh.

The `macostools` module defines the following functions:

copy *(src, dst[, createpath])* -- function of module macostools

> Copy file *src* to *dst*. The files can be specified as pathnames or `FSSpec` objects. If *createpath* is non-zero *dst* must be a pathname and the folders leading to the destination are created if necessary. The method copies data and resource fork and some finder information (creator, type and flags). Custom icons, comments and icon position are not copied.

copytree *(src, dst)* -- function of module macostools

> Recursively copy a file tree from *src* to *dst*, creating folders as needed. *Src* and *dst* should be specified as pathnames.

mkalias *(src, dst)* -- function of module macostools

> Create a finder alias *dst* pointing to *src*. Both may be specified as pathnames or *FSSpec* objects.

BUFSIZ -- data of module macostools

> The buffer size for `copy`, default 1 megabyte.

Note that the process of creating finder aliases is not specified in the Apple documentation. Hence, aliases created with `mkalias` could conceivably have incompatible behaviour in some cases.

Next: mactcp Prev: MacOS Up: Macintosh Specific Services Top: Top

Profiler Introduction -- Python library reference

10.1. Introduction to the profiler

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

AL (uppercase) -- Python library reference

Next: cd Prev: al Up: SGI IRIX Specific Services Top: Top

16.2. 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 *

Next: cd Prev: al Up: SGI IRIX Specific Services Top: Top

mimetools -- Python library reference

11.11. Standard Module mimetools

This module defines a subclass of the class `rfc822.Message` and a number of utility functions that are useful for the manipulation for MIME style multipart or encoded message.

It defines the following items:

Message *(fp)* -- function of module mimetools

> Return a new instance of the `mimetools.Message` class. This is a subclass of the `rfc822.Message` class, with some additional methods (see below).

choose_boundary *()* -- function of module mimetools

> Return a unique string that has a high likelihood of being usable as a part boundary. The string has the form `"hostipaddr.uid.pid.timestamp.random"`.

decode *(input, output, encoding)* -- function of module mimetools

> Read data encoded using the allowed MIME *encoding* from open file object *input* and write the decoded data to open file object *output*. Valid values for *encoding* include `"base64"`, `"quoted-printable"` and `"uuencode"`.

encode *(input, output, encoding)* -- function of module mimetools

> Read data from open file object *input* and write it encoded using the allowed MIME *encoding* to open file object *output*. Valid values for *encoding* are the same as for `decode()`.

copyliteral *(input, output)* -- function of module mimetools

> Read lines until EOF from open file *input* and write them to open file *output*.

copybinary *(input, output)* -- function of module mimetools

> Read blocks until EOF from open file *input* and write them to open file *output*. The block size is currently fixed at 8192.

Menu

regsub -- Python library reference

4.3. 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)* -- function of module regsub

> 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)* -- function of module regsub

> 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)* -- function of module regsub

> 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']`.

termios -- Python library reference

Next: TERMIOS Prev: gdbm Up: UNIX Specific Services Top: Top

8.7. Built-in Module termios

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

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

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

The module defines the following functions:

tcgetattr *(fd)* -- function of module termios

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

tcsetattr *(fd, when, attributes)* -- function of module termios

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

tcsendbreak *(fd, duration)* -- function of module termios

> Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25--0.5 seconds; a nonzero *duration* has a system dependent meaning.

tcdrain *(fd)* -- function of module termios

> Wait until all output written to file descriptor *fd* has been transmitted.

tcflush *(fd, queue)* -- function of module termios

> Discard queued data on file descriptor *fd*. The *queue* selector specifies which queue: `TERMIOS.TCIFLUSH` for the input queue, `TERMIOS.TCOFLUSH` for the output queue, or `TERMIOS.TCIOFLUSH` for both queues.

tcflow *(fd, action)* -- function of module termios

> Suspend or resume input or output on file descriptor *fd*. The *action* argument can be `TERMIOS.TCOOFF` to suspend output, `TERMIOS.TCOON` to restart output, `TERMIOS.TCIOFF` to suspend input, or `TERMIOS.TCION` to restart input.

Menu

Next: TERMIOS Prev: gdbm Up: UNIX Specific Services Top: Top

fcntl -- Python library reference

8.9. Built-in Module fcntl

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

The module defines the following functions:

fcntl *(fd, op[, arg])* -- function of module struct

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

ioctl *(fd, op, arg)* -- function of module struct

> This function is identical to the `fcntl()` function, except that the operations are typically defined in the library module `IOCTL`.

If the library modules `FCNTL` or `IOCTL` are missing, you can find the opcodes in the C include files `sys/fcntl` and `sys/ioctl`. You can create the modules yourself with the h2py script, found in the `Tools/scripts` directory. Examples (all on a SVR4 compliant system):

```
import struct, FCNTL
```

file = open(...)

rv = fcntl(file.fileno(), FCNTL.O_NDELAY, 1)

lockdata = struct.pack('hhllhh', FCNTL.F_WRLCK, 0, 0, 0, 0, 0)

rv = fcntl(file.fileno(), FCNTL.F_SETLKW, lockdata)

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

OldProfile Class -- Python library reference

Next: HotProfile Class Prev: Profiler Extensions Up: Profiler Extensions Top: Top

10.8.1. OldProfile Class

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

```
class OldProfile(Profile):



    def trace_dispatch_exception(self, frame, t):

        rt, rtt, rct, rfn, rframe, rcur = self.cur

        if rcur and not rframe is frame:

            return self.trace_dispatch_return(rframe, t)

        return 0



    def trace_dispatch_call(self, frame, t):

        fn = `frame.f_code`



        self.cur = (t, 0, 0, fn, frame, self.cur)

        if self.timings.has_key(fn):

            tt, ct, callers = self.timings[fn]

            self.timings[fn] = tt, ct, callers
```

```
        else:
            self.timings[fn] = 0, 0, {}
        return 1

    def trace_dispatch_return(self, frame, t):
        rt, rtt, rct, rfn, frame, rcur = self.cur
        rtt = rtt + t
        sft = rtt + rct

        pt, ptt, pct, pfn, pframe, pcur = rcur
        self.cur = pt, ptt+rt, pct+sft, pfn, pframe, pcur

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

```
        return 1

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

macfs -- Python library reference

14.6. Built-in Module macfs

This module provides access to macintosh FSSpec handling, the Alias Manager, finder aliases and the Standard File package.

Whenever a function or method expects a *file* argument, this argument can be one of three things: (1) a full or partial Macintosh pathname, (2) an FSSpec object or (3) a 3-tuple `(wdRefNum, parID, name)` as described in Inside Mac VI. and the standard file package can also be found there.

FSSpec *(file)* -- function of module macfs

> Create an FSSpec object for the specified file.

RawFSSpec *(data)* -- function of module macfs

> Create an FSSpec object given the raw data for the C structure for the FSSpec as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

RawAlias *(data)* -- function of module macfs

> Create an Alias object given the raw data for the C structure for the alias as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

FInfo *()* -- function of module macfs

> Create a zero-filled FInfo object.

ResolveAliasFile *(file)* -- function of module macfs

> Resolve an alias file. Returns a 3-tuple *`(fsspec, isfolder, aliased)`* where *fsspec* is the resulting FSSpec object, *isfolder* is true if *fsspec* points to a folder and *aliased* is true if the file was an alias in the first place (otherwise the FSSpec object for the file itself is returned).

StandardGetFile *([type, ...])* -- function of module macfs

> Present the user with a standard ``open input file'' dialog. Optionally, you can pass up to four 4-char file types to limit the files the user can choose from. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

PromptGetFile *(prompt[, type, ...])* -- function of module macfs

> Similar to *StandardGetFile* but allows you to specify a prompt.

StandardPutFile *(prompt, [default])* -- function of module macfs

> Present the user with a standard ``open output file'' dialog. *prompt* is the prompt string, and the optional *default* argument initializes the output file name. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

GetDirectory *([prompt])* -- function of module macfs

> Present the user with a non-standard ``select a directory'' dialog. *prompt* is the prompt string, and the optional. Return an FSSpec object and a success-indicator.

FindFolder *(where, which, create)* -- function of module macfs

> Locates one of the ``special'' folders that MacOS knows about, such as the trash or the Preferences folder. *Where* is the disk to search, *which* is the 4-char string specifying which folder to locate. Setting *create* causes the folder to be created if it does not exist. Returns a `(vrefnum, dirid)` tuple.

The constants for *where* and *which* can be obtained from the standard module *MACFS*.

Menu

struct -- Python library reference

4.4. 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.

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

error -- exception of module struct

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

pack *(fmt, v1, v2, ...)* -- function of module struct

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)* -- function of module struct

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)* -- function of module struct

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.)

Prev: regsub Up: String Services Top: Top

File Objects -- Python library reference

2.1.7.8. 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. They are also returned by some other built-in functions and methods, e.g. `posix.popen()` and `posix.fdopen()` and the `makefile()` method of socket objects. 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 *()* -- Method on file

> Close the file. A closed file cannot be read or written anymore.

flush *()* -- Method on file

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

isatty *()* -- Method on file

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

read *([size])* -- Method on file

> 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 negative or 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 *([size])* -- Method on file

> Read one entire line from the file. A trailing newline character is kept in the string(1) (but may be absent when a file ends with an incomplete line). If thevarsize argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. 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 *()* -- Method on file

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

seek *(offset, whence)* -- Method on file

> 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 *()* -- Method on file

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

write *(str)* -- Method on file

> Write a string to the file. There is no return value.

writelines *(list)* -- Method on file

> Write a list of strings to the file. There is no return value. (The name is intended to match

`readlines`; `writelines` does not add line separators.)

---------- Footnotes ----------

(1) The advantage of leaving the newline on is that an empty string can be returned to mean EOF without being ambiguous. Another advantage is that (in cases where it might matter, e.g. if you want to make an exact copy of a file while scanning its lines) you can tell whether the last line of a file ended in a newline or not (yes this happens!).

Next: Internal Objects Prev: The Null Object Up: Other Built-in Types Top: Top

Unresolved Jumps

The following references where not loaded when creating this help file

../dir.html

Unavailable reference...

The reference `../dir.html` was not included in this help file
Details: (2, 'No such file or directory')