Macintosh Library Modules

Release 1.5.2

Guido van Rossum

March 22, 2000

Corporation for National Research Initiatives
1895 Preston White Drive, Reston, VA 20191, USA
E-mail: guido@python.org

Abstract

This library reference manual documents Python's extensions for the Macintosh. It should be used in conjunction with the *Python Library Reference*, which documents the standard library and built-in types.

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

Contents

1 Introduction

The modules in this manual are available on the Apple Macintosh only.

Aside from the modules described here there are also interfaces to various MacOS toolboxes, which are currently not extensively described. The toolboxes for which modules exist are: `AE` (Apple Events), `Cm` (Component Manager), `Ctl` (Control Manager), `Dlg` (Dialog Manager), `Evt` (Event Manager), `Fm` (Font Manager), `List` (List Manager), `Menu` (Moenu Manager), `Qd` (QuickDraw), `Qt` (QuickTime), `Res` (Resource Manager and Handles), `Scrap` (Scrap Manager), `Snd` (Sound Manager), `TE` (TextEdit), `Waste` (non-Apple **TextEdit** replacement) and `Win` (Window Manager).

If applicable the module will define a number of Python objects for the various structures declared by the toolbox, and operations will be implemented as methods of the object. Other operations will be implemented as functions in the module. Not all operations possible in C will also be possible in Python (callbacks are often a problem), and parameters will occasionally be different in Python (input and output buffers, especially). All methods and functions have a `__doc__` string describing their arguments and return values, and for additional description you are referred to *Inside Macintosh* or similar works.

The following modules are documented here:

`mac`	Implementations for the `os` module.
`macpath`	MacOS path manipulation functions.
`ctb`	Interfaces to the Communications Tool Box. Only the Connection Manager is supported.
`macconsole`	Think C's console package.
`macdnr`	Interfaces to the Macintosh Domain Name Resolver.
`macfs`	Support for FSSpec, the Alias Manager, **finder** aliases, and the Standard File package.
`ic`	Access to Internet Config.
`MacOS`	Access to MacOS specific interpreter features.
`macostools`	Convenience routines for file manipulation.
`findertools`	Wrappers around the **finder**'s Apple Events interface.
`mactcp`	The MacTCP interfaces.
`macspeech`	Interface to the Macintosh Speech Manager.
`EasyDialogs`	Basic Macintosh dialogs.
`FrameWork`	Interactive application framework.
`MiniAEFrame`	Support to act as an Open Scripting Architecture (OSA) server ("Apple Events").
`MiniAEFrame`	Support to act as an Open Scripting Architecture (OSA) server ("Apple Events").

2 `mac` — Implementations for the `os` module

This module implements the operating system dependent functionality provided by the standard module `os`. It is best accessed through the `os` module.

The following functions are available in this module: `chdir()`, `close()`, `dup()`, `fdopen()`, `getcwd()`, `lseek()`, `listdir()`, `mkdir()`, `open()`, `read()`, `rename()`, `rmdir()`, `stat()`, `sync()`, `unlink()`, `write()`, as well as the exception `error`. Note that the times returned by `stat()` are floating-point values, like all time values in MacPython.

One additional function is available:

`xstat(`*path*`)`
: This function returns the same information as `stat()`, but with three additional values appended: the size of the resource fork of the file and its 4-character creator and type.

3 `macpath` — MacOS path manipulation functions

This module is the Macintosh implementation of the `os.path` module. It is most portably accessed as `os.path`. Refer to the *Python Library Reference* for documentation of `os.path`.

The following functions are available in this module: `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. For other functions available in `os.path` dummy counterparts are available.

4 ctb — Interface to the Communications Tool Box

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

`error`
: The exception raised on errors.

`cmData`
`cmCntl`
`cmAttn`
: Flags for the *channel* argument of the `Read()` and `Write()` methods.

`cmFlagsEOM`
: End-of-message flag for `Read()` and `Write()`.

`choose*`
: Values returned by `Choose()`.

`cmStatus*`
: Bits in the status as returned by `Status()`.

`available()`
: Return `1` if the Communication Toolbox is available, zero otherwise.

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

4.1 Connection Objects

For all connection methods that take a *timeout* argument, a value of `-1` is indefinite, meaning that the command runs to completion.

`callback`
: If this member is set to a value other than `None` it should point to a function accepting a single argument (the connection object). This will make all connection object methods work asynchronously, with the callback routine being called upon completion.

  Note: for reasons beyond my understanding the callback routine is currently never called. You are advised against using asynchronous calls for the time being.

`Open(`*timeout*`)`
: Open an outgoing connection, waiting at most *timeout* seconds for the connection to be established.

`Listen(`*timeout*`)`
: Wait for an incoming connection. Stop waiting after *timeout* seconds. This call is only meaningful to some tools.

`accept(`*yesno*`)`
: Accept (when *yesno* is non-zero) or reject an incoming call after `Listen()` returned.

`Close(`*timeout, now*`)`
: Close a connection. When *now* is zero, the close is orderly (i.e. outstanding output is flushed, etc.)

with a timeout of *timeout* seconds. When *now* is non-zero the close is immediate, discarding output.

`Read(`*len, chan, timeout*`)`
Read *len* bytes, or until *timeout* seconds have passed, from the channel *chan* (which is one of `cmData`, `cmCntl` or `cmAttn`). Return a 2-tuple: the data read and the end-of-message flag, `cmFlagsEOM`.

`Write(`*buf, chan, timeout, eom*`)`
Write *buf* to channel *chan*, aborting after *timeout* seconds. When *eom* has the value `cmFlagsEOM`, an end-of-message indicator will be written after the data (if this concept has a meaning for this communication tool). The method returns the number of bytes written.

`Status()`
Return connection status as the 2-tuple `(`*sizes*`, `*flags*`)`. *sizes* is a 6-tuple giving the actual buffer sizes used (see `CMNew()`), *flags* is a set of bits describing the state of the connection.

`GetConfig()`
Return the configuration string of the communication tool. These configuration strings are tool-dependent, but usually easily parsed and modified.

`SetConfig(`*str*`)`
Set the configuration string for the tool. The strings are parsed left-to-right, with later values taking precedence. This means individual configuration parameters can be modified by simply appending something like `'baud 4800'` to the end of the string returned by `GetConfig()` and passing that to this method. The method returns the number of characters actually parsed by the tool before it encountered an error (or completed successfully).

`Choose()`
Present the user with a dialog to choose a communication tool and configure it. If there is an outstanding connection some choices (like selecting a different tool) may cause the connection to be aborted. The return value (one of the `choose*` constants) will indicate this.

`Idle()`
Give the tool a chance to use the processor. You should call this method regularly.

`Abort()`
Abort an outstanding asynchronous `Open()` or `Listen()`.

`Reset()`
Reset a connection. Exact meaning depends on the tool.

`Break(`*length*`)`
Send a break. Whether this means anything, what it means and interpretation of the *length* parameter depends on the tool in use.

5 `macconsole` — Think C's console package

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

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

`C_ECHO`
`C_NOECHO`
`C_CBREAK`
`C_RAW`
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()`
Open a new console window. Return a console window object.

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

5.1 macconsole options object

These options are examined when a window is created:

`top`
`left`
The origin of the window.

`nrows`
`ncols`
The size of the window.

`txFont`
`txSize`
`txStyle`
The font, fontsize and fontstyle to be used in the window.

`title`
The title of the window.

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

5.2 console window object

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

`setmode(`*mode*`)`
Set the input mode of the console to `C_ECHO`, etc.

`settabs(`*n*`)`
Set the tabsize to *n* spaces.

`cleos()`
Clear to end-of-screen.

`cleol()`
Clear to end-of-line.

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

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

`hide()`
Hide the window, remembering the contents.

`show()`
Show the window again.

`echo2printer()`
: Copy everything written to the window to the printer as well.

6 `macdnr` — Interface to the Macintosh Domain Name Resolver

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

`Close()`
: Close the resolver extension. Again, not needed for normal use.

`StrToAddr(`*hostname*`)`
: Look up the IP address for *hostname*. This call returns a dnr result object of the "address" variation.

`AddrToName(`*addr*`)`
: Do a reverse lookup on the 32-bit integer IP-address *addr*. Returns a dnr result object of the "address" variation.

`AddrToStr(`*addr*`)`
: Convert the 32-bit integer IP-address *addr* to a dotted-decimal string. Returns the string.

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

`MXInfo(`*domain*`)`
: Query the nameservers for a mail exchanger for *domain*. This is the hostname of a host willing to accept SMTP mail for the given domain. Returns a dnr result object of the "mx" variety.

6.1 DNR Result Objects

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

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

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

`wait()`
: Wait for the query to complete.

`isdone()`
: Return `1` if the query is complete.

`rtnCode`
: The error code returned by the query.

`cname`
The canonical name of the host that was queried.

`ip0`
`ip1`
`ip2`
`ip3`
At most four integer IP addresses for this host. Unused entries are zero. Valid only for address queries.

`cpuType`
`osType`
Textual strings giving the machine type an OS name. Valid for "hinfo" queries.

`exchange`
The name of a mail-exchanger host. Valid for "mx" queries.

`preference`
The preference of this mx record. Not too useful, since the Macintosh will only return a single mx record. Valid for "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)
```

7 `macfs` — Various file system services

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 Macintosh: Files*. A description of aliases and the Standard File package can also be found there.

`FSSpec(`*file*`)`
Create an FSSpec object for the specified file.

`RawFSSpec(`*data*`)`
Create an FSSpec object given the raw data for the C structure for the FSSpec as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

`RawAlias(`*data*`)`
Create an Alias object given the raw data for the C structure for the alias as a string. This is mainly useful if you have obtained an FSSpec structure over a network.

`FInfo()`
Create a zero-filled FInfo object.

`ResolveAliasFile(`*file*`)`
Resolve an alias file. Returns a 3-tuple (*fsspec*, *isfolder*, *aliased*) where *fsspec* is the resulting FSSpec object, *isfolder* is true if *fsspec* points to a folder and *aliased* is true if the file was an alias in the first place (otherwise the FSSpec object for the file itself is returned).

`StandardGetFile(`[*type, ...*]`)`
Present the user with a standard "open input file" dialog. Optionally, you can pass up to four 4-character 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, ...*]`)`
Similar to `StandardGetFile()` but allows you to specify a prompt.

`StandardPutFile(`*prompt,* [*default*]`)`
Present the user with a standard "open output file" dialog. *prompt* is the prompt string, and the optional *default* argument initializes the output file name. The function returns an FSSpec object and a flag indicating that the user completed the dialog without cancelling.

`GetDirectory(`[*prompt*]`)`
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.

`SetFolder(`[*fsspec*]`)`
Set the folder that is initially presented to the user when one of the file selection dialogs is presented. *fsspec* should point to a file in the folder, not the folder itself (the file need not exist, though). If no argument is passed the folder will be set to the current directory, i.e. what `os.getcwd()` returns.

Note that starting with system 7.5 the user can change Standard File behaviour with the "general controls" controlpanel, thereby making this call inoperative.

`FindFolder(`*where, which, create*`)`
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-character string specifying which folder to locate. Setting *create* causes the folder to be created if it does not exist. Returns a `(`*vrefnum*`,` *dirid*`)` tuple.

`NewAliasMinimalFromFullPath(`*pathname*`)`
Return a minimal alias object that points to the given file, which must be specified as a full pathname. This is the only way to create an Alias pointing to a non-existing file.

The constants for *where* and *which* can be obtained from the standard module *MACFS*.

`FindApplication(`*creator*`)`
Locate the application with 4-char creator code *creator*. The function returns an FSSpec object pointing to the application.

7.1 FSSpec objects

`data`
The raw data from the FSSpec object, suitable for passing to other applications, for instance.

`as_pathname()`
Return the full pathname of the file described by the FSSpec object.

`as_tuple()`
Return the `(`*wdRefNum*`,` *parID*`,` *name*`)` tuple of the file described by the FSSpec object.

`NewAlias(`[*file*]`)`
Create an Alias object pointing to the file described by this FSSpec. If the optional *file* parameter is present the alias will be relative to that file, otherwise it will be absolute.

`NewAliasMinimal()`
Create a minimal alias pointing to this file.

`GetCreatorType()`
Return the 4-character creator and type of the file.

`SetCreatorType(`*creator, type*`)`
Set the 4-character creator and type of the file.

`GetFInfo()`
Return a FInfo object describing the finder info for the file.

`SetFInfo(`*finfo*`)`
Set the finder info for the file to the values given as *finfo* (an FInfo object).

`GetDates()`
Return a tuple with three floating point values representing the creation date, modification date and backup date of the file.

`SetDates(`*crdate, moddate, backupdate*`)`
Set the creation, modification and backup date of the file. The values are in the standard floating point format used for times throughout Python.

7.2 Alias Objects

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

`Resolve(`$\left[$*file*$\right]$`)`
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. If the file does not exist but the path leading up to it does exist a valid fsspec is returned.

`GetInfo(`*num*`)`
An interface to the C routine `GetAliasInfo()`.

`Update(`*file,* $\left[$*file2*$\right]$`)`
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` value from the Alias object and modifying the resource.

7.3 FInfo Objects

See *Inside Macintosh: Files* for a complete description of what the various fields mean.

`Creator`
The 4-character creator code of the file.

`Type`
The 4-character type code of the file.

`Flags`
The finder flags for the file as 16-bit integer. The bit values in *Flags* are defined in standard module `MACFS`.

`Location`
A Point giving the position of the file's icon in its folder.

`Fldr`
The folder the file is in (as an integer).

8 ic — Access to Internet Config

This module provides access to Macintosh Internet Config package, which stores preferences for Internet programs such as mail address, default homepage, etc. Also, Internet Config contains an elaborate set of mappings from Macintosh creator/type codes to foreign filename extensions plus information on how to transfer files (binary, ascii, etc).

There is a low-level companion module `icglue` which provides the basic Internet Config access functionality. This low-level module is not documented, but the docstrings of the routines document the parameters and the routine names are the same as for the Pascal or C API to Internet Config, so the standard IC programmers' documentation can be used if this module is needed.

The `ic` module defines the `error` exception and symbolic names for all error codes Internet Config can produce; see the source for details.

`error`
: Exception raised on errors in the `ic` module.

The `ic` module defines the following class and function:

`IC(`[*signature*[, *ic*]]`)`
: Create an internet config object. The signature is a 4-character creator code of the current application (default `'Pyth'`) which may influence some of ICs settings. The optional *ic* argument is a low-level `icglue.icinstance` created beforehand, this may be useful if you want to get preferences from a different config file, etc.

`launchurl(`*url*[, *hint*]`)`
`parseurl(`*data*[, *start*[, *end*[, *hint*]]]`)`
`mapfile(`*file*`)`
`maptypecreator(`*type, creator*[, *filename*]`)`
`settypecreator(`*file*`)`
: These functions are "shortcuts" to the methods of the same name, described below.

8.1 IC Objects

`IC` objects have a mapping interface, hence to obtain the mail address you simply get *ic*`['MailAddress']`. Assignment also works, and changes the option in the configuration file.

The module knows about various datatypes, and converts the internal IC representation to a "logical" Python data structure. Running the `ic` module standalone will run a test program that lists all keys and values in your IC database, this will have to server as documentation.

If the module does not know how to represent the data it returns an instance of the `ICOpaqueData` type, with the raw data in its `data` attribute. Objects of this type are also acceptable values for assignment.

Besides the dictionary interface, `IC` objects have the following methods:

`launchurl(`*url*[, *hint*]`)`
: Parse the given URL, lauch the correct application and pass it the URL. The optional *hint* can be a scheme name such as `'mailto:'`, in which case incomplete URLs are completed with this scheme. If *hint* is not provided, incomplete URLs are invalid.

`parseurl(`*data*[, *start*[, *end*[, *hint*]]]`)`
: Find an URL somewhere in *data* and return start position, end position and the URL. The optional *start* and *end* can be used to limit the search, so for instance if a user clicks in a long textfield you can pass the whole textfield and the click-position in *start* and this routine will return the whole URL in which the user clicked. As above, *hint* is an optional scheme used to complete incomplete URLs.

`mapfile(`*file*`)`
: Return the mapping entry for the given *file*, which can be passed as either a filename or an `macfs.FSSpec()` result, and which need not exist.

  The mapping entry is returned as a tuple (*version*, *type*, *creator*, *postcreator*, *flags*, *extension*, *appname*, *postappname*, *mimetype*, *entryname*), where *version* is the entry version number, *type* is the 4-character filetype, *creator* is the 4-character creator type, *postcreator* is the 4-character creator code of an optional application to post-process the file after downloading, *flags* are various bits specifying whether to transfer in binary or ascii and such, *extension* is the filename extension for this file type, *appname* is the printable name of the application to which this file belongs, *postappname* is the name of the postprocessing application, *mimetype* is the MIME type of this file and *entryname* is the name of this entry.

`maptypecreator(`*type, creator*[, *filename*]`)`
: Return the mapping entry for files with given 4-character *type* and *creator* codes. The optional *filename* may be specified to further help finding the correct entry (if the creator code is `'????'`, for instance).

  The mapping entry is returned in the same format as for *mapfile*.

`settypecreator(`*file*`)`
: Given an existing *file*, specified either as a filename or as an `macfs.FSSpec()` result, set its creator and type correctly based on its extension. The finder is told about the change, so the finder icon will be updated quickly.

9 MacOS — Access to MacOS interpreter features

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 artifact.

`Error`
: 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. Symbolic names for all known error codes are defined in the standard module `macerrors`.

`SetEventHandler(`*handler*`)`
: In the inner interpreter loop Python will occasionally check for events, unless disabled with `ScheduleParams()`. With this function you can pass a Python event-handler function that will be called if an event is available. The event is passed as parameter and the function should return non-zero if the event has been fully processed, otherwise event processing continues (by passing the event to the console window package, for instance).

  Call `SetEventHandler()` without a parameter to clear the event handler. Setting an event handler while one is already set is an error.

`SchedParams(`[*doint*[, *evtmask*[, *besocial*[, *interval*[, *bgyield*]]]]]`)`
: Influence the interpreter inner loop event handling. *Interval* specifies how often (in seconds, floating point) the interpreter should enter the event processing code. When true, *doint* causes interrupt (command-dot) checking to be done. *evtmask* tells the interpreter to do event processing for events in the mask (redraws, mouseclicks to switch to other applications, etc). The *besocial* flag gives other processes a chance to run. They are granted minimal runtime when Python is in the foreground and *bgyield* seconds per *interval* when Python runs in the background.

  All parameters are optional, and default to the current value. The return value of this function is a tuple with the old values of these options. Initial defaults are that all processing is enabled, checking is done every quarter second and the CPU is given up for a quarter second when in the background.

`HandleEvent(`*ev*`)`
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.

If you attempt to call this function from an event handler set through `SetEventHandler()` you will get an exception.

`GetErrorString(`*errno*`)`
Return the textual description of MacOS error code *errno*.

`splash(`*resid*`)`
This function will put a splash window on-screen, with the contents of the DLOG resource specified by *resid*. Calling with a zero argument will remove the splash screen. This function is useful if you want an applet to post a splash screen early in initialization without first having to load numerous extension modules.

`DebugStr(`*message* [, *object*]`)`
Drop to the low-level debugger with message *message*. The optional *object* argument is not used, but can easily be inspected from the debugger.

Note that you should use this function with extreme care: if no low-level debugger like MacsBug is installed this call will crash your system. It is intended mainly for developers of Python extension modules.

`openrf(`*name* [, *mode*]`)`
Open the resource fork of a file. Arguments are the same as for the built-in function `open()`. The object returned has file-like semantics, but it is not a Python file object, so there may be subtle differences.

10 `macostools` — Convenience routines for file manipulation

This module contains some convenience routines for file-manipulation on the Macintosh.

The `macostools` module defines the following functions:

`copy(`*src, dst*[, *createpath*[, *copytimes*]]`)`
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, flags) and optionally the creation, modification and backup times (default is to copy them). Custom icons, comments and icon position are not copied.

If the source is an alias the original to which the alias points is copied, not the aliasfile.

`copytree(`*src, dst*`)`
Recursively copy a file tree from *src* to *dst*, creating folders as needed. *src* and *dst* should be specified as pathnames.

`mkalias(`*src, dst*`)`
Create a finder alias *dst* pointing to *src*. Both may be specified as pathnames or FSSpec objects.

`touched(`*dst*`)`
Tell the finder that some bits of finder-information such as creator or type for file *dst* has changed. The file can be specified by pathname or fsspec. This call should prod the finder into redrawing the files icon.

`BUFSIZ`
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.

11 `findertools` — The **finder**'s Apple Events interface

This module contains routines that give Python programs access to some functionality provided by the finder. They are implemented as wrappers around the AppleEvent interface to the finder.

All file and folder parameters can be specified either as full pathnames or as FSSpec objects.

The `findertools` module defines the following functions:

`launch(`*file*`)`
: Tell the finder to launch *file*. What launching means depends on the file: applications are started, folders are opened and documents are opened in the correct application.

`Print(`*file*`)`
: Tell the finder to print a file (again specified by full pathname or FSSpec). The behaviour is identical to selecting the file and using the print command in the finder.

`copy(`*file, destdir*`)`
: Tell the finder to copy a file or folder *file* to folder *destdir*. The function returns an Alias object pointing to the new file.

`move(`*file, destdir*`)`
: Tell the finder to move a file or folder *file* to folder *destdir*. The function returns an Alias object pointing to the new file.

`sleep()`
: Tell the finder to put the Macintosh to sleep, if your machine supports it.

`restart()`
: Tell the finder to perform an orderly restart of the machine.

`shutdown()`
: Tell the finder to perform an orderly shutdown of the machine.

12 `mactcp` — The MacTCP interfaces

This module provides an interface to the Macintosh TCP/IP driver MacTCP. There is an accompanying module, `macdnr`, which provides an interface to the name-server (allowing you to translate hostnames to IP addresses), a module `MACTCPconst` which has symbolic names for constants constants used by MacTCP. Since the built-in module `socket` is also available on the Macintosh it is usually easier to use sockets instead of the Macintosh-specific MacTCP API.

A complete description of the MacTCP interface can be found in the Apple MacTCP API documentation.

`MTU()`
: Return the Maximum Transmit Unit (the packet size) of the network interface.

`IPAddr()`
: Return the 32-bit integer IP address of the network interface.

`NetMask()`
: Return the 32-bit integer network mask of the interface.

`TCPCreate(`*size*`)`
: Create a TCP Stream object. *size* is the size of the receive buffer, `4096` is suggested by various sources.

`UDPCreate(`*size, port*`)`

Create a UDP Stream object. *size* is the size of the receive buffer (and, hence, the size of the biggest datagram you can receive on this port). *port* is the UDP port number you want to receive datagrams on, a value of zero will make MacTCP select a free port.

12.1 TCP Stream Objects

`asr`
: When set to a value different than `None` this should refer 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. Macintosh documentation calls this the *asynchronous service routine*. 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 *asr*.

`PassiveOpen(`*port*`)`
: Wait for an incoming connection on TCP port *port* (zero makes the system pick a free port). The call returns immediately, and you should use `wait()` to wait for completion. You should not issue any method calls other than `wait()`, `isdone()` or `GetSockName()` before the call completes.

`wait()`
: Wait for `PassiveOpen()` to complete.

`isdone()`
: Return `1` if a `PassiveOpen()` has completed.

`GetSockName()`
: Return the TCP address of this side of a connection as a 2-tuple (*host*, *port*), both integers.

`ActiveOpen(`*lport, host, rport*`)`
: Open an outgoing connection to TCP address (*host*, *rport*). Use local port *lport* (zero makes the system pick a free port). This call blocks until the connection has been established.

`Send(`*buf, push, urgent*`)`
: Send data *buf* over the connection. *push* and *urgent* are flags as specified by the TCP standard.

`Rcv(`*timeout*`)`
: Receive data. The call returns when *timeout* seconds have passed or when (according to the MacTCP documentation) "a reasonable amount of data has been received". The return value is a 3-tuple (*data*, *urgent*, *mark*). If urgent data is outstanding `Rcv` will always return that before looking at any normal data. The first call returning urgent data will have the *urgent* flag set, the last will have the *mark* flag set.

`Close()`
: Tell MacTCP that no more data will be transmitted on this connection. The call returns when all data has been acknowledged by the receiving side.

`Abort()`
: Forcibly close both sides of a connection, ignoring outstanding data.

`Status()`
: Return a TCP status object for this stream giving the current status (see below).

12.2 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`

`localPort`
`remoteHost`
`remotePort`
: The integer IP-addresses and port numbers of both endpoints of the connection.

`sendWindow`
: The current window size.

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

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

12.3 UDP Stream Objects

Note that, unlike the name suggests, there is nothing stream-like about UDP.

`asr`
: The asynchronous service routine to be called on events such as datagram arrival without outstanding `Read` call. The *asr* has a single argument, the event code.

`port`
: A read-only member giving the port number of this UDP Stream.

`Read(`*timeout*`)`
: Read a datagram, waiting at most *timeout* seconds (-1 is infinite). Return the data.

`Write(`*host, port, buf*`)`
: Send *buf* as a datagram to IP-address *host*, port *port*.

13 `macspeech` — Interface to the Macintosh Speech Manager

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

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

`Version()`
: Return the (integer) version number of the Speech Manager.

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

`Busy()`
: Return the number of speech channels busy, system-wide.

`CountVoices()`
: Return the number of different voices available.

`GetIndVoice(`*num*`)`
: Return a Voice object for voice number *num*.

13.1 Voice Objects

Voice objects contain the description of a voice. It is currently not yet possible to access the parameters of a voice.

`GetGender()`
Return the gender of the voice: `0` for male, `1` for female and `-1` for neuter.

`NewChannel()`
Return a new Speech Channel object using this voice.

13.2 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*`)`
Start uttering the given string.

`Stop()`
Stop babbling.

`GetPitch()`
Return the current pitch of the channel, as a floating-point number.

`SetPitch(`*pitch*`)`
Set the pitch of the channel.

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

`SetRate(`*rate*`)`
Set the speech rate of the channel.

14 EasyDialogs — Basic Macintosh dialogs

The `EasyDialogs` module contains some simple dialogs for the Macintosh, modelled after the `stdwin` dialogs with similar names. All routines have an optional parameter *id* with which you can override the DLOG resource used for the dialog, as long as the item numbers correspond. See the source for details.

The `EasyDialogs` module defines the following functions:

`Message(`*str*`)`
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*$\left[$*, default*$\right]$`)`
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*$\left[$*, default*$\right]$`)`
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.

`ProgressBar(`$\left[$*label*$\left[$*, maxval*$\right]\right]$`)`

Display a modeless progress dialog with a thermometer bar. *label* is the text string displayed (default "Working..."), *maxval* is the value at which progress is complete (default `100`). The returned object has one method, `set(`*value*`)`, which sets the value of the progress bar. The bar remains visible until the object returned is discarded.

The progress bar has a "cancel" button, but it is currently non-functional.

Note that `EasyDialogs` does not currently use the notification manager. This means that displaying dialogs while the program is in the background will lead to unexpected results and possibly crashes. Also, all dialogs are modeless and hence expect to be at the top of the stacking order. This is true when the dialogs are created, but windows that pop-up later (like a console window) may also result in crashes.

15 `FrameWork` — Interactive application 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 or the examples for more details.

The `FrameWork` module defines the following functions:

`Application()`
: 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()`
: An object representing the menubar. This object is usually not created by the user.

`Menu(`*bar, title*[*, after*]`)`
: 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*]`)`
: 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.

  In stead of a callable object the callback can also be a string. In this case menu selection causes the lookup of a method in the topmost window and the application. The method name is the callback string with `'domenu_'` prepended.

  Calling the `MenuBar fixmenudimstate()` method sets the correct dimming for all menu items based on the current front window.

`Separator(`*menu*`)`
: Add a separator to the end of a menu.

`SubMenu(`*menu, label*`)`
: Create a submenu named *label* under menu *menu*. The menu object is returned.

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

`DialogWindow(`*parent*`)`

Creates a modeless dialog window.

`windowbounds(`*width, height*`)`
Return a (*left*, *top*, *right*, *bottom*) tuple suitable for creation of a window of given width and height. The window will be staggered with respect to previous windows, and an attempt is made to keep the whole window on-screen. The window will however always be exact the size given, so parts may be offscreen.

`setwatchcursor()`
Set the mouse cursor to a watch.

`setarrowcursor()`
Set the mouse cursor to an arrow.

15.1 Application Objects

Application objects have the following methods, among others:

`makeusermenus()`
Override this method if you need menus in your application. Append the menus to the attribute `menubar`.

`getabouttext()`
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*]]`)`
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). While raising *self* to exit the mainloop is still supported it is not recommended: call `self._quit()` instead.

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.

In general, all event handlers should return `1` if the event is fully handled and `0` otherwise (because the front window was not a FrameWork window, for instance). This is needed so that update events and such can be passed on to other windows like the Sioux console window. Calling `MacOS.HandleEvent()` is not allowed within *our_dispatch* or its callees, since this may result in an infinite loop if the code is called through the Python inner-loop event handler.

`asyncevents(`*onoff*`)`
Call this method with a nonzero parameter to enable asynchronous event handling. This will tell the inner interpreter loop to call the application event handler *async_dispatch* whenever events are available. This will cause FrameWork window updates and the user interface to remain working during long computations, but will slow the interpreter down and may cause surprising results in non-reentrant code (such as FrameWork itself). By default *async_dispatch* will immedeately call *our_dispatch* but you may override this to handle only certain events asynchronously. Events you do not handle will be passed to Sioux and such.

The old on/off value is returned.

`_quit()`
Terminate the running `mainloop()` call at the next convenient moment.

`do_char(`*c, event*`)`
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*`)`
: 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).

`idle(`*event*`)`
: Called by the main event loop when no events are available. The null-event is passed (so you can look at mouse position, etc).

15.2 Window Objects

Window objects have the following methods, among others:

`open()`
: Override this method to open a window. Store the MacOS window-id in `self.wid` and call the `do_postopen()` method to register the window with the parent application.

`close()`
: Override this method to do any special processing on window close. Call the `do_postclose()` method to cleanup the parent state.

`do_postresize(`*width, height, macoswindowid*`)`
: Called after the window is resized. Override if more needs to be done than calling `InvalRect`.

`do_contentclick(`*local, modifiers, event*`)`
: The user clicked in the content part of a window. The arguments are the coordinates (window-relative), the key modifiers and the raw event.

`do_update(`*macoswindowid, event*`)`
: An update event for the window was received. Redraw the window.

`do_activate(`*activate, event*`)`
: The window was activated (*activate* `==` `1`) or deactivated (*activate* `==` `0`). Handle things like focus highlighting, etc.

15.3 ControlsWindow Object

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

`do_controlhit(`*window, control, pcode, event*`)`
: Part *pcode* of control *control* was hit by the user. Tracking and such has already been taken care of.

15.4 ScrolledWindow Object

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

`scrollbars(`$\left[$*wantx*$\left[$`,` *wanty*$\right]\right]$`)`
: Create (or destroy) horizontal and vertical scrollbars. The arguments specify which you want (default: both). The scrollbars always have minimum `0` and maximum `32767`.

`getscrollbarvalues()`
: You must supply this method. It should return a tuple `(`*x*`,` *y*`)` giving the current position of the scrollbars (between `0` and `32767`). You can return `None` for either to indicate the whole document is visible in that direction.

`updatescrollbars()`
: Call this method when the document has changed. It will call `getscrollbarvalues()` and update the

scrollbars.

`scrollbar_callback(`*which, what, value*`)`
: Supplied by you and called after user interaction. *which* will be `'x'` or `'y'`, *what* will be `'-'`, `'--'`, `'set'`, `'++'` or `'+'`. For `'set'`, *value* will contain the new scrollbar position.

`scalebarvalues(`*absmin, absmax, curmin, curmax*`)`
: Auxiliary method to help you calculate values to return from `getscrollbarvalues()`. You pass document minimum and maximum value and topmost (leftmost) and bottommost (rightmost) visible values and it returns the correct number or `None`.

`do_activate(`*onoff, event*`)`
: Takes care of dimming/highlighting scrollbars when a window becomes frontmost vv. If you override this method call this one at the end of your method.

`do_postresize(`*width, height, window*`)`
: Moves scrollbars to the correct position. Call this method initially if you override it.

`do_controlhit(`*window, control, pcode, event*`)`
: Handles scrollbar interaction. If you override it call this method first, a nonzero return value indicates the hit was in the scrollbars and has been handled.

15.5 DialogWindow Objects

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

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

`do_itemhit(`*item, event*`)`
: Item number *item* was hit. You are responsible for redrawing toggle buttons, etc.

16 `MiniAEFrame` — Open Scripting Architecture server support

The module `MiniAEFrame` provides a framework for an application that can function as an Open Scripting Architecture (OSA) server, i.e. receive and process AppleEvents. It can be used in conjunction with `FrameWork` or standalone.

This module is temporary, it will eventually be replaced by a module that handles argument names better and possibly automates making your application scriptable.

The `MiniAEFrame` module defines the following classes:

`AEServer()`
: A class that handles AppleEvent dispatch. Your application should subclass this class together with either `MiniApplication` or `FrameWork.Application`. Your `__init__()` method should call the `__init__()` method for both classes.

`MiniApplication()`
: A class that is more or less compatible with `FrameWork.Application` but with less functionality. Its event loop supports the apple menu, command-dot and AppleEvents; other events are passed on to the Python interpreter and/or Sioux. Useful if your application wants to use `AEServer` but does not provide its own windows, etc.

16.1 AEServer Objects

`installaehandler`(*classe, type, callback*)
: Installs an AppleEvent handler. *classe* and *type* are the four-character OSA Class and Type designators, `'****'` wildcards are allowed. When a matching AppleEvent is received the parameters are decoded and your callback is invoked.

`callback`(*_object, **kwargs*)
: Your callback is called with the OSA Direct Object as first positional parameter. The other parameters are passed as keyword arguments, with the 4-character designator as name. Three extra keyword parameters are passed: `_class` and `_type` are the Class and Type designators and `_attributes` is a dictionary with the AppleEvent attributes.

  The return value of your method is packed with `aetools.packevent()` and sent as reply.

Note that there are some serious problems with the current design. AppleEvents which have non-identifier 4-character designators for arguments are not implementable, and it is not possible to return an error to the originator. This will be addressed in a future release.

Module Index

C

E

F

I

M

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X