Macintosh Library Modules

Release 2.2.2

Guido van Rossum
Fred L. Drake, Jr., editor

October 14, 2002

PythonLabs
Email: python-docs@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

CHAPTER

ONE

Using Python on the Macintosh

Using Python on the Macintosh can seem like something completely different than using it on a UNIX-like or Windows system. Most of the Python documentation, both the "official" documentation and published books, describe only how Python is used on these systems, causing confusion for the new user of MacPython. This chapter gives a brief introduction to the specifics of using Python on a Macintosh.

1.1 Getting and Installing MacPython

The most recent release version as well as possible newer experimental versions are best found at the MacPython page maintained by Jack Jansen: http://www.cwi.nl/ jack/macpython.html.

Please refer to the 'README' included with your distribution for the most up-to-date instructions.

1.2 Entering the interactive Interpreter

The interactive interpreter that you will see used in Python documentation is started by double-clicking the **PythonInterpreter** icon, which looks like a 16-ton weight falling. You should see the version information and the '>>> ' prompt. Use it exactly as described in the standard documentation.

1.3 How to run a Python script

There are several ways to run an existing Python script; two common ways to run a Python script are "drag and drop" and "double clicking". Other ways include running it from within the IDE (see Section 1.6), or launching via AppleScript.

1.3.1 Drag and drop

One of the easiest ways to launch a Python script is via "Drag and Drop". This is just like launching a text file in the Finder by "dragging" it over your word processor's icon and "dropping" it there. Make sure that you use an icon referring to the **PythonInterpreter**, not the **IDE** or **Idle** icons which have different behaviour which is described below.

Some things that might have gone wrong:

- A window flashes after dropping the script onto the **PythonInterpreter**, but then disappears. Most likely this is a configuration issue; your **PythonInterpreter** is setup to exit immediately upon completion, but your script assumes that if it prints something that text will stick around for a while. To fix this, see section 1.7.3.

- After dropping the script onto the **PythonInterpreter**, a window appeared which said: "File contains `\r` characters (incorrect line endings?)". That script probably originated on a UNIX or Windows machine. You will need to change the line endings to the standard Mac usage. One way to do this is to open the file in **BBedit** (http://www.barebones.com/products/bbedit_lite.html) which can easily change the line endings between Mac, DOS, and UNIX styles.
- When you waved the script icon over the **PythonInterpreter**, the **PythonInterpreter** icon did not hilight. Most likely the Creator code and document type is unset (or set incorrectly) – this often happens when a file originates on a non-Mac computer. See section 1.3.2 for more details.

1.3.2 Set Creator and Double Click

If the script that you want to launch has the appropriate Creator Code and File Type you can simply double-click on the script to launch it. To be "double-clickable" a file needs to be of type '`TEXT`', with a creator code of '`Pyth`'.

Setting the creator code and filetype can be done with the IDE (see sections 1.6.2 and 1.6.4), with an editor with a Python mode (**BBEdit**) – see section 1.5.1, or with assorted other Mac utilities, but a script ('fixfiletypes.py') has been included in the MacPython distribution, making it possible to set the proper Type and Creator Codes with Python.

The 'fixfiletypes.py' script will change the file type and creator codes for the indicated directory. To use 'fixfiletypes.py':

1. Locate it in the 'scripts' folder of the 'Mac' folder of the MacPython distribution.
2. Put all of the scripts that you want to fix in a folder with nothing else in it.
3. Double-click on the 'fixfiletypes.py' icon.
4. Navigate into the folder of files you want to fix, and press the "Select current folder" button.

1.4 Simulating command line arguments

There are two ways to simulate command-line arguments with MacPython.

1. via Interpreter options
   - Hold the option-key down when launching your script. This will bring up a dialog box of Python Interpreter options.
   - Click "Set UNIX-style command line.." button.
   - Type the arguments into the "Argument" field.
   - Click "OK"
   - Click "Run".
2. via drag and drop If you save the script as an applet (see Section 1.6.4), you can also simulate some command-line arguments via "Drag-and-Drop". In this case, the names of the files that were dropped onto the applet will be appended to `sys.argv`, so that it will appear to the script as though they had been typed on a command line. As on UNIX systems, the first item in `sys.srgv` is the path to the applet, and the rest are the files dropped on the applet.

1.5 Creating a Python script

Since Python scripts are simply text files, they can be created in any way that text files can be created, but some special tools also exist with extra features.

1.5.1 In an editor

You can create a text file with any word processing program such as **MSWord** or **AppleWorks** but you need to make sure that the file is saved as "ASCII" or "plain text".

Editors with Python modes

Several text editors have additional features that add functionality when you are creating a Python script. These can include coloring Python keywords to make your code easier to read, module browsing, or a built-in debugger. These include **Alpha**, **Pepper**, and **BBedit**, and the MacPython IDE (Section 1.6).

BBedit

If you use **BBEdit** to create your scripts you will want to tell it about the Python creator code so that you can simply double click on the saved file to launch it.

- Launch **BBEdit**.
- Select "Preferences" from the "Edit" menu.
- Select "File Types" from the scrolling list.
- click on the "Add..." button and navigate to **PythonInterpreter** in the main directory of the MacPython distribution; click "open".
- Click on the "Save" button in the Preferences panel.

1.6 The IDE

The **Python IDE** (Integrated Development Environment) is a separate application that acts as a text editor for your Python code, a class browser, a graphical debugger, and more.

1.6.1 Using the "Python Interactive" window

Use this window like you would the **PythonInterpreter**, except that you cannot use the "Drag and drop" method above. Instead, dropping a script onto the **Python IDE** icon will open the file in a separate script window (which you can then execute manually – see section 1.6.3).

1.6.2 Writing a Python Script

In addition to using the **Python IDE** interactively, you can also type out a complete Python program, saving it incrementally, and execute it or smaller selections of it.

You can create a new script, open a previously saved script, and save your currently open script by selecting the appropriate item in the "File" menu. Dropping a Python script onto the **Python IDE** will open it for editting.

If you try to open a script with the **Python IDE** but either can't locate it from the "Open" dialog box, or you get an error message like "Can't open file of type ..." see section 1.3.2.

When the **Python IDE** saves a script, it uses the creator code settings which are available by clicking on the small black triangle on the top right of the document window, and selecting "save options". The default is to save the file with the **Python IDE** as the creator, this means that you can open the file for editing by simply double-clicking on its icon. You might want to change this behaviour so that it will be opened by the **PythonInterpreter**, and run. To do this simply choose "Python Interpreter" from the "save options". Note that these options are associated with the *file* not the application.

1.6.3 Executing a script from within the IDE

You can run the script in the frontmost window of the **Python IDE** by hitting the run all button. You should be aware, however that if you use the Python convention '`if __name__ == "__main__":`' the script will *not* be "__main__" by default. To get that behaviour you must select the "Run as __main__" option from the small black triangle on the top right of the document window. Note that this option is associated with the *file* not the application. It *will* stay active after a save, however; to shut this feature off simply select it again.

1.6.4 "Save as" versus "Save as Applet"

When you are done writing your Python script you have the option of saving it as an "applet" (by selecting "Save as applet" from the "File" menu). This has a significant advantage in that you can drop files or folders onto it, to pass them to the applet the way command-line users would type them onto the command-line to pass them as arguments to the script. However, you should make sure to save the applet as a separate file, do not overwrite the script you are writing, because you will not be able to edit it again.

Accessing the items passed to the applet via "drag-and-drop" is done using the standard `sys.argv` mechanism. See the general documentation for more

Note that saving a script as an applet will not make it runnable on a system without a Python installation.

1.7 Configuration

The MacPython distribution comes with **EditPythonPrefs**, an applet which will help you to customize the MacPython environment for your working habits.

1.7.1 EditPythonPrefs

EditPythonPrefs gives you the capability to configure Python to behave the way you want it to. There are two ways to use **EditPythonPrefs**, you can use it to set the preferences in general, or you can drop a particular Python engine onto it to customize only that version. The latter can be handy if, for example, you want to have a second copy of the **PythonInterpreter** that keeps the output window open on a normal exit even though you prefer to normally not work that way.

To change the default preferences, simply double-click on **EditPythonPrefs**. To change the preferences only for one copy of the Interpreter, drop the icon for that copy onto **EditPythonPrefs**. You can also use **EditPythonPrefs** in this fashion to set the preferences of the **Python IDE** and any applets you create – see section 1.6.4.

1.7.2 Adding modules to the Module Search Path

When executing an `import` statement, Python looks for modules in places defined by the `sys.path` To edit the `sys.path` on a Mac, launch **EditPythonPrefs**, and enter them into the largish field at the top (one per line).

Since MacPython defines a main Python directory, the easiest thing is to add folders to search within the main Python directory. To add a folder of scripts that you created called "My Folder" located in the main Python Folder, enter '`$(PYTHON):My Folder`' onto a new line.

To add the Desktop under OS 9 or below, add '`StartupDriveName:Desktop Folder`' on a new line.

1.7.3 Default startup options

The “Default startup options...” button in the **EditPythonPrefs** dialog box gives you many options including the ability to keep the “Output” window open after the script terminates, and the ability to enter interactive mode after the termination of the run script. The latter can be very helpful if you want to examine the objects that were created during your script.

1.8 Mac OS X

At the time of this writing Mac OS X had just been released as a Public Beta. Efforts are under way to bring MacPython to Mac OS X. The MacPython release 2.2.21.5.2c1 runs quite well within the “Classic” environment. A “Carbon” port of the MacPython code is being prepared for release, and several people have made a command line version available to the “Darwin” layer (which is accessible via Terminal.app).

CHAPTER

TWO

MacPython Modules

The following modules are only available on the Macintosh, and 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.
macfs	Support for FSSpec, the Alias Manager, **finder** aliases, and the Standard File package.
ic	Access to Internet Config.
MacOS	Access to Mac OS-specific interpreter features.
macostools	Convenience routines for file manipulation.
findertools	Wrappers around the **finder**'s Apple Events interface.
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").
aepack	Conversion between Python variables and AppleEvent data containers.
aetypes	Python representation of the Apple Event Object Model.

2.1 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, but only under Classic MacPython, not under Carbon MacPython:

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. Availability: Classic MacPython only. **Deprecated since release 2.3.** Use the `macfs.FSSpec()` function to get an FSSpec object for the file, then use the `GetCreatorType()` method to get the creator and type information. It will no longer be possible to get the size of the resource fork.

2.2 `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.

2.3 `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.

This module is only available under MacOS9 or earlier, in classic PPC MacPython.

`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.

2.3.1 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 (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.

2.4 `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.

Note: A module, `macfsn`, is auto-imported to replace StandardFile calls in `macfs` with NavServices calls.

`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 which will be displayed at the top of the dialog.

`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. You have to first open the directory before clicking on the "select current directory" button. *prompt* is the prompt string which will be displayed at the top of the dialog. 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" control panel, 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-character creator code *creator*. The function returns an FSSpec object pointing to the application.

2.4.1 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.

2.4.2 Alias Objects

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

Resolve([*file*])
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*[, *file2*])
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.

2.4.3 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).

2.5 `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.). Since MacOS 9, this module is a control panel named Internet.

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.

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

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

class `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.

2.5.1 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 serve 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 text field you can pass the whole text field 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.

2.6 `MacOS` — Access to Mac OS 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 capitalization of the module name; this is a historical artifact.

`runtimemodel`
Either `'ppc'`, `'carbon'` or `'macho'`. This signifies whether this Python uses the classic (InterfaceLib style) runtime model, the Mac OS X compatible CarbonLib style or the Mac OS X-only Mach-O style.

exception `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 `Sched-`

`uleParams()`. 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 processor 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.

2.7 `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.

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 tell the finder to redraw 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.

2.8 `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's file menu.

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.

2.9 `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.

This module is only available on MacOS9 and earlier in classic PPC MacPython.

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*.

2.9.1 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.

2.9.2 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.

2.10 `EasyDialogs` — Basic Macintosh dialogs

The `EasyDialogs` module contains some simple dialogs for the Macintosh. All routines take an optional resource ID parameter *id* with which one can override the `DLOG` resource used for the dialog, provided that the dialog items correspond (both type and item number) to those in the default `DLOG` resource. See source code for details.

The `EasyDialogs` module defines the following functions:

`Message`(*str*[, *id*[, *ok=None*]])
Displays a modal dialog with the message text *str*, which should be at most 255 characters long. The button text defaults to "OK", but is set to the string argument *ok* if the latter is supplied. Control is returned when the user clicks the "OK" button.

`AskString`(*prompt*[, *default*[, *id*[, *ok*[, *cancel*]]]])
Asks the user to input a string value via a modal dialog. *prompt* is the prompt message, and the optional *default* supplies the initial value for the string (otherwise `""` is used). The text of the "OK" and "Cancel" buttons can be changed with the *ok* and *cancel* arguments. All strings can be at most 255 bytes long. `AskString()` returns the string entered or `None` in case the user cancelled.

`AskPassword`(*prompt*[, *default*[, *id*[, *ok*[, *cancel*]]]])
Asks the user to input a string value via a modal dialog. Like `AskString()`, but with the text shown as bullets. The arguments have the same meaning as for `AskString()`.

`AskYesNoCancel`(*question*[, *default*[, *yes*[, *no*[, *cancel*[, *id*]]]]])
Presents a dialog with prompt *question* and three buttons labelled "Yes", "No", and "Cancel". Returns `1` for "Yes", `0` for "No" and `-1` for "Cancel". The value of *default* (or `0` if *default* is not supplied) is returned when the `RETURN` key is pressed. The text of the buttons can be changed with the *yes*, *no*, and *cancel* arguments; to prevent a button from appearing, supply `""` for the corresponding argument.

`ProgressBar`([*title*[, *maxval*[, *label*[, *id*]]]])
Displays a modeless progress-bar dialog. This is the constructor for the `ProgressBar` class described below. *title* is the text string displayed (default "Working..."), *maxval* is the value at which progress is complete (default `0`, indicating that an indeterminate amount of work remains to be done), and *label* is the text that is displayed above the progress bar itself.

`GetArgv`([*optionlist*[*commandlist*[, *addoldfile*[, *addnewfile*[, *addfolder*[, *id*]]]]]])
Displays a dialog which aids the user in constructing a command-line argument list. Returns the list in `sys.argv` format, suitable for passing as an argument to `getopt.getopt()`. *addoldfile*, *addnewfile*, and *addfolder* are boolean arguments. When nonzero, they enable the user to insert into the command line paths to an existing file, a (possibly) not-yet-existent file, and a folder, respectively. (Note: Option arguments must appear in the command line before file and folder arguments in order to be recognized by `getopt.getopt()`.) Arguments containing spaces can be specified by enclosing them within single or double quotes. A `SystemExit` exception is raised if the user presses the "Cancel" button.

optionlist is a list that determines a popup menu from which the allowed options are selected. Its items can take one of two forms: *optstr* or `(`*optstr*`, `*descr*`)`. When present, *descr* is a short descriptive string that is displayed in the dialog while this option is selected in the popup menu. The correspondence between *optstr*s and command-line arguments is:

***optstr* format**	**Command-line format**
`x`	**-x** (short option)
`x:` or `x=`	**-x** (short option with value)
`xyz`	**--xyz** (long option)
`xyz:` or `xyz=`	**--xyz** (long option with value)

commandlist is a list of items of the form *cmdstr* or `(`*cmdstr*`, `*descr*`)`, where *descr* is as above. The *cmdstr*s will appear in a popup menu. When chosen, the text of *cmdstr* will be appended to the command line as is, except that a trailing '`:`' or '`=`' (if present) will be trimmed off.

New in version 2.0.

2.10.1 ProgressBar Objects

`ProgressBar` objects provide support for modeless progress-bar dialogs. Both determinate (thermometer style) and indeterminate (barber-pole style) progress bars are supported. The bar will be determinate if its maximum value is greater than zero; otherwise it will be indeterminate. Changed in version 2.2: Support for indeterminate-style progress bars was added.

The dialog is displayed immediately after creation. If the dialog's "Cancel" button is pressed, or if `Cmd-.` or `ESC` is typed, the dialog window is hidden and `KeyboardInterrupt` is raised (but note that this response does not occur until the progress bar is next updated, typically via a call to `inc()` or `set()`). Otherwise, the bar remains visible until the `ProgressBar` object is discarded.

`ProgressBar` objects possess the following attributes and methods:

`curval`
: The current value (of type integer or long integer) of the progress bar. The normal access methods coerce `curval` between `0` and `maxval`. This attribute should not be altered directly.

`maxval`
: The maximum value (of type integer or long integer) of the progress bar; the progress bar (thermometer style) is full when `curval` equals `maxval`. If `maxval` is `0`, the bar will be indeterminate (barber-pole). This attribute should not be altered directly.

`title`(*[newstr]*)
: Sets the text in the title bar of the progress dialog to *newstr*.

`label`(*[newstr]*)
: Sets the text in the progress box of the progress dialog to *newstr*.

`set`(*value[, max]*)
: Sets the progress bar's `curval` to *value*, and also `maxval` to *max* if the latter is provided. *value* is first coerced between 0 and `maxval`. The thermometer bar is updated to reflect the changes, including a change from indeterminate to determinate or vice versa.

`inc`(*[n]*)
: Increments the progress bar's `curval` by *n*, or by `1` if *n* is not provided. (Note that *n* may be negative, in which case the effect is a decrement.) The progress bar is updated to reflect the change. If the bar is indeterminate, this causes one "spin" of the barber pole. The resulting `curval` is coerced between 0 and `maxval` if incrementing causes it to fall outside this range.

2.11 `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 following are some comments posted on the MacPython newsgroup about the strengths and limitations of `FrameWork`:

> The strong point of `FrameWork` is that it allows you to break into the control-flow at many different places. `W`, for instance, uses a different way to enable/disable menus and that plugs right in leaving the rest intact. The weak points of `FrameWork` are that it has no abstract command interface (but that shouldn't be difficult), that it's dialog support is minimal and that it's control/toolbar support is non-existent.

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 create, the item 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.

Instead 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. However, the window will however always be the exact size given, so parts may be offscreen.

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

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

2.11.1 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).

2.11.2 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.

2.11.3 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.

2.11.4 ScrolledWindow Object

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

scrollbars([*wantx*[, *wanty*]])
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. 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.

2.11.5 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.

2.12 `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. As an example, it is used in **PythonCGISlave**.

The `MiniAEFrame` module defines the following classes:

class `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.

class `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.

2.12.1 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.

2.13 `aepack` — Conversion between Python variables and AppleEvent data containers

The `aepack` module defines functions for converting (packing) Python variables to AppleEvent descriptors and back (unpacking). Within Python the AppleEvent descriptor is handled by Python objects of built-in type AEDesc, defined in module `AE`.

The `aepack` module defines the following functions:

`pack``(`*x*[, *forcetype*]`)`
: Returns an `AEDesc` object containing a conversion of Python value `x`. If *forcetype* is provided it specifies the descriptor type of the result. Otherwise, a default mapping of Python types to Apple Event descriptor types is used, as follows:

Python type	descriptor type
`FSSpec`	typeFSS
`Alias`	typeAlias
integer	typeLong (32 bit integer)
float	typeFloat (64 bit floating point)
string	typeText
list	typeAEList
dictionary	typeAERecord
instance	*see below*

FSSpec and Alias are built-in object types defined in the module `macfs`.

If *x* is a Python instance then this function attempts to call an `__aepack__()` method. This method should return an AE.AEDesc object.

If the conversion *x* is not defined above, this function returns the Python string representation of a value (the repr() function) encoded as a text descriptor.

`unpack`(*x*)

x must be an object of type `AEDesc`. This function returns a Python object representation of the data in the Apple Event descriptor *x*. Simple AppleEvent data types (integer, text, float) are returned as their obvious Python counterparts. Apple Event lists are returned as Python lists, and the list elements are recursively unpacked. Object references (ex. `line 3 of document 1`) are returned as instances of `aetypes.ObjectSpecifier`. AppleEvent descriptors with descriptor type typeFSS are returned as `FSSpec` objects. AppleEvent record descriptors are returned as Python dictionaries, with keys of type `?` and elements recursively unpacked.

See Also:

Module `AE` (section **??**):
Built-in access to Apple Event Manager routines.

Module `aetypes` (section 2.14):
Python definitions of codes for Apple Event descriptor types.

Inside Macintosh: Interapplication Communication
(http://developer.apple.com/techpubs/mac/IAC/IAC-2.html)
Information about inter-process communications on the Macintosh.

2.14 `aetypes` — AppleEvent objects

The `aetypes` defines classes used to represent Apple Event object specifiers. An object specifier is essentially an address of an object implemented in a Apple Event server. An Apple Event specifier is used as the direct object for an Apple Event or as the argument of an optional parameter. In AppleScript an object specifier is represented by a phrase such as: `character 23 of document "Semprini"`. The classes defined in this module allow this specifier to be represented by a Python object which is initialized as follows: `res = Document(1).Character(23)`

The `AEObjects` module defines the following class:

class `ObjectSpecifier`(*want, form, seld, from*)

This is the base class for representing object specifiers and is generally not constructed directly by the user. Its important functionality is to define an `__aepack__()` function, which returns the Apple Event descriptor containing the object specifier. Its data members, set directly from the constructor arguments, are:

`want`

A four character string representing the class code of the object. These class codes are specified in Apple Event Suites; for example the standard code for a character object is the 4 bytes '`char`'.

CHAPTER

THREE

MacOS Toolbox Modules

There are a set of modules that provide interfaces to various MacOS toolboxes. 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.

These modules all live in a package called `Carbon`. Despite the name Carbon they are also available under classic PPC MacPython. The normal use pattern is

```
from Carbon import AE
```

Warning! These modules are not yet documented. If you wish to contribute documentation of any of these modules, please get in touch with python-docs@python.org.

Carbon.AE	Interface to the Apple Events toolbox
Carbon.App	Interface to the Appearance Manager
Carbon.CF	Interface to the Core Foundation
Carbon.Cm	Interface to the Component Manager
Carbon.Ctl	Interface to the Control Manager
Carbon.Dlg	Interface to the Dialog Manager
Carbon.Evt	Interface to the Event Manager
Carbon.Fm	Interface to the Font Manager
Carbon.Help	Interface to the Balloon Help Manager
Carbon.List	Interface to the List Manager
Carbon.Menu	Interface to the Menu Manager
Carbon.Mlte	Interface to the MultiLingual Text Editor
Carbon.Qd	Interface to the QuickDraw toolbox
Carbon.Qdoffs	Interface to the QuickDraw Offscreen APIs
Carbon.Qt	Interface to the QuickTime toolbox
Carbon.Res	Interface to the Resource Manager and Handles
Carbon.Scrap	The Scrap Manager provides basic services for implementing cut & paste and clipboard operations.
Carbon.Snd	Interface to the Sound Manager
Carbon.TE	Interface to TextEdit
Carbon.Win	Interface to the Window Manager
ColorPicker	Interface to the standard color selection dialog.

3.1 `Carbon.AE` — Apple Events

3.2 `Carbon.App` — Appearance Manager

3.3 `Carbon.CF` — Core Foundation

This module is only available under Carbon MacPython. The `CFBase`, `CFArray`, `CFData`, `CFDictionary`, `CFString` and `CFURL` objects are supported, some only partially.

3.4 `Carbon.Cm` — Component Manager

3.5 `Carbon.Ctl` — Control Manager

3.6 `Carbon.Dlg` — Dialog Manager

3.7 `Carbon.Evt` — Event Manager

3.8 `Carbon.Fm` — Font Manager

3.9 `Carbon.Help` — Help Manager

This module is only available under MacOS9 and earlier in classic PPC MacPython.

3.10 `Carbon.List` — List Manager

3.11 `Carbon.Menu` — Menu Manager

3.12 `Carbon.Mlte` — MultiLingual Text Editor

3.13 `Carbon.Qd` — QuickDraw

3.14 `Carbon.Qdoffs` — QuickDraw Offscreen

3.15 `Carbon.Qt` — QuickTime

3.16 `Carbon.Res` — Resource Manager and Handles

3.17 `Carbon.Scrap` — Scrap Manager

This module is only fully available on MacOS9 and earlier under classic PPC MacPython. Very limited functionality is available under Carbon MacPython.

See Also:

Scrap Manager
(http://developer.apple.com/techpubs/mac/MoreToolbox/MoreToolbox-109.html)
: Apple's documentation for the Scrap Manager gives a lot of useful information about using the Scrap Manager in applications.

3.18 `Carbon.Snd` — Sound Manager

3.19 `Carbon.TE` — TextEdit

3.20 `Carbon.Win` — Window Manager

3.21 `ColorPicker` — Color selection dialog

The `ColorPicker` module provides access to the standard color picker dialog.

`GetColor`(*prompt, rgb*)
: Show a standard color selection dialog and allow the user to select a color. The user is given instruction by the *prompt* string, and the default color is set to *rgb*. *rgb* must be a tuple giving the red, green, and blue components of the color. `GetColor()` returns a tuple giving the user's selected color and a flag indicating whether they accepted the selection of cancelled.

CHAPTER

FOUR

Undocumented Modules

The modules in this chapter are poorly documented (if at all). If you wish to contribute documentation of any of these modules, please get in touch with python-docs@python.org.

applesingle	Rudimentary decoder for AppleSingle format files
buildtools	Helper module for BuildApplet, BuildApplication and macfreeze
py_resource	Helper to create `'PYC '` resources for compiled applications
cfmfile	Code Fragment Resource module
icopen	Internet Config replacement for `open()`
macerrors	Constant definitions for many Mac OS error codes
macfsn	NavServices versions of StandardFile calls
macresource	Locate script resources
Nac	Interface to Navigation Services
mactty	Easy access serial to line connections
mkcwproject	Create CodeWarrior projects
nsremote	Wrapper around Netscape OSA modules
PixMapWrapper	Wrapper for PixMap objects
preferences	Nice application preferences manager with support for defaults
pythonprefs	Specialized preferences manager for the Python interpreter
quietconsole	buffered, non-visible stdout output
videoreader	read QuickTime movies frame by frame for further processing
W	Widgets for the Mac, built on top of `FrameWork`
waste	Interface to the "WorldScript-Aware Styled Text Engine."

4.1 `applesingle` — AppleSingle decoder

4.2 `buildtools` — Helper module for BuildApplet and Friends

4.3 `py_resource` — Resources from Python code

This module is primarily used as a help module for **BuildApplet** and **BuildApplication**. It is able to store compiled Python code as `'PYC '` resources in a file.

4.4 `cfmfile` — Code Fragment Resource module

`cfmfile` is a module that understands Code Fragments and the accompanying "cfrg" resources. It can parse them and merge them, and is used by BuildApplication to combine all plugin modules to a single executable.

4.5 `icopen` — Internet Config replacement for `open()`

Importing `icopen` will replace the builtin `open()` with a version that uses Internet Config to set file type and creator for new files.

4.6 `macerrors` — Mac OS Errors

`macerrors` cotains constant definitions for many Mac OS error codes.

4.7 `macfsn` — NavServices calls

`macfsn` contains wrapper functions that have the same API as the macfs StandardFile calls, but are implemented with Navigation Services through the `Nav` module. Importing it will replace the methods in `macfs` with these, if Navigation Services is available on your machine.

4.8 `macresource` — Locate script resources

`macresource` helps scripts finding their resources, such as dialogs and menus, without requiring special case code for when the script is run under MacPython, as a MacPython applet or under OSX Python.

4.9 `Nav` — NavServices calls

A low-level interface to Navigation Services.

4.10 `mactty` — Serial line connections

This module is only available under Mac OS 9 or earlier in classic PPC MacPython.

4.11 `mkcwproject` — Create CodeWarrior projects

`mkcwproject` creates project files for the Metrowerks CodeWarrior development environment. It is a helper module for `distutils` but can be used separately for more control.

4.12 `nsremote` — Wrapper around Netscape OSA modules

`nsremote` is a wrapper around the Netscape OSA modules that allows you to easily send your browser to a given URL. A related module that may be of interest is the `webbrowser` module, documented in the *Python Library Reference*.

4.13 `PixMapWrapper` — Wrapper for PixMap objects

`PixMapWrapper` wraps a PixMap object with a Python object that allows access to the fields by name. It also has methods to convert to and from `PIL` images.

4.14 `preferences` — Application preferences manager

The `preferences` module allows storage of user preferences in the system-wide preferences folder, with defaults coming from the application itself and the possibility to override preferences for specific situations.

4.15 `pythonprefs` — Preferences manager for Python

This module is a specialization of the `preferences` module that allows reading and writing of the preferences for the Python interpreter.

4.16 `quietconsole` — non-visible stdout output

`quietconsole` allows you to keep stdio output in a buffer without displaying it (or without displaying the stdout window altogether, if set with **EditPythonPrefs**) until you try to read from stdin or disable the buffering, at which point all the saved output is sent to the window. Good for programs with graphical user interfaces that do want to display their output at a crash.

4.17 `videoreader` — read QuickTime movies

`videoreader` reads and decodes QuickTime movies and passes a stream of images to your program. It also provides some support for audio tracks.

4.18 `W` — Widgets built on `FrameWork`

The `W` widgets are used extensively in the **IDE**.

4.19 `waste` — non-Apple **TextEdit** replacement

See Also:

About WASTE
(http://www.merzwaren.com/waste/)
Information about the WASTE widget and library, including documentation and downloads.

APPENDIX

A

History and License

A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Zope Corporation (then Digital Creations; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

A.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.2.2 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.2.2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001, 2002 Python Software Foundation; All Rights Reserved" are retained in Python 2.2.2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.2.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.2.2.

4. PSF is making Python 2.2.2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.2.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.2.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.2.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.2.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N