

Script Debugger Help



Welcome to Script Debugger, from [Late Night Software](#)!



Script Debugger is a powerful environment for easy, rapid development of AppleScript solutions.

You'll discover that Script Debugger is the best way to:

- **Explore** scriptable applications.
 - [Examine](#), [navigate](#) and [search](#) an application's [dictionary](#).
 - [Probe](#) a running application's scriptable objects and values.
- **Develop** AppleScript code.
 - Script Debugger helps you [edit code](#) and [insert](#) boilerplate constructs.
 - [Run](#) your code, examine the [result](#), and view the [Apple events](#) that you are sending and receiving.
 - Run your code [a little at a time](#) to better understand how it works (or to figure out why it doesn't).

About This Help Document:

Use the hyperlinks, and the navigation aids at the top and bottom of each page, to learn about Script Debugger or to reach the information you need.

This help document is searchable (using Help Viewer).

For a complete **Table of Contents**, [click here](#).

Or, click the blue arrow in the upper right corner repeatedly, to read every page in order, like a book!

Further Details:

[Opening and Saving Scripts](#)
[Explore](#)

[Develop](#)
[Reference](#)



Opening and Saving Scripts



A script is a file consisting of AppleScript code. Scripts are Script Debugger's native documents. Read on to learn how Script Debugger opens and saves scripts.

Learn how Script Debugger:

- [Opens scripts](#). What file formats can Script Debugger open? What happens if there's difficulty opening a file?
- [Saves scripts](#). What formats does Script Debugger save in? What additional information does Script Debugger save?

Also, Script Debugger gives you access to secondary information about a script file. Learn about:

- [Description](#). A script's description can serve as a reminder to the developer, an explanation to users, and a splash screen in an applet.
- [Manifest](#). Script Debugger creates a summary of what applications and scripting additions are needed in order for your script to be edited and executed.
- [Library](#). A script can depend on the loading of code from other scripts. Script Debugger automates this aspect of AppleScript.

Further Details:

[Open](#)
[Save](#)
[Description](#)
[Library](#)
[Manifest](#)





Here's how to open a file with Script Debugger.

To **make a new script**, choose File > New. (Alternatively, there are various ways that you can make a new script [targeting a particular application](#).)

Here's how to set the [default appearance and features](#) of a new script.

To **open an existing script**, do any of the following:

- Choose File > Open.
- If you've recently had this script open, choose it from File > Recent Scripts.
- Drag and drop the script file onto Script Debugger's icon in the Finder or the Dock.
- If the script's [owner](#) is Script Debugger, double-click it in the Finder. (But that doesn't work for an [applet](#), since by default when an applet is opened from the Finder, it runs.)

A [General preference](#) lets you tell Script Debugger to [warn you](#) if opening an existing script might cause an application to [launch](#).

Learn [what files Script Debugger can open](#).

Learn how Script Debugger can help when there's [trouble opening a file](#).

Further Details:

[Compatibility](#)
[Opening a Compiled Script as Text](#)



Compatibility



Script Debugger can open any of AppleScript's native file types from any period in AppleScript's history. This includes:

- **Compiled script file with the bytecode in the resource fork.** This is the oldest format, going back as far as AppleScript itself.
- **Compiled script file with the bytecode in the data fork.** This is the default format created by the current version of Apple's Script Editor.
- **Script bundle.** This is a [bundle](#) with the bytecode as a [data fork](#) compiled script file inside the bundle. It is compatible with Panther (Mac OS X 10.3) and later.
- **Applet.** Script Debugger can open [applets](#) created on any system.
- **Applet bundle.** This is a [bundle applet](#) format compatible with Panther (Mac OS X 10.3) and later.

When Script Debugger opens a script, it [retains the current values](#) of the script's top-level entities (such as properties).

Script Debugger can also open a **text file**. If the file starts with a [UTF-8 or UTF-16 Unicode BOM](#) (byte order mark), it will be interpreted accordingly; otherwise, the file is assumed to be in the [MacRoman](#) encoding. [Line endings](#) can be Mac or Unix.

Script Debugger can also [save](#) in a full range of formats.

[Opening a Compiled Script as Text](#)





Opening a Compiled Script as Text



Sometimes, AppleScript prevents Script Debugger from opening a [compiled script file](#), or shows the file's contents with raw Apple event codes. This indicates that something has gone wrong with the [decompilation](#) process. For example, an application or scripting addition [needed by the script](#) is missing, or a script's internal alias to an application has broken.

If the script was originally saved with Script Debugger, you can **open the script as text**. To do so:

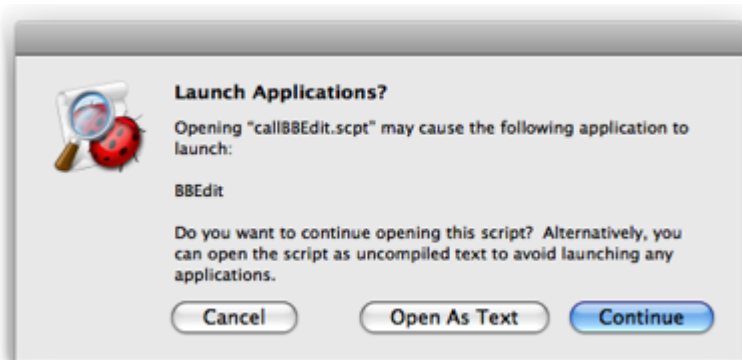
- Choose File > Recover Script.

This feature works because when Script Debugger saves a compiled script, it saves not only the compiled [bytecode](#) but also the uncompiled text. The uncompiled text is placed in the file's [resource fork](#) (or, if the file is a [bundle](#), in a file within the bundle).

Warning: You can accidentally strip away a [compiled script's resource fork](#). You might, for example, save the compiled script file on a non-HFS filing system (true Unix, or Windows), or send it through email without compressing it, or open it with some badly behaved script editor application. Also, if you edit a script with some other script editor application, the stored uncompiled text may no longer match the current state of the bytecode. If that happens, the original text is gone, and if there is then a problem with opening the compiled script file, this feature *won't* work (Script Debugger won't be able to recover the original text).

Script Debugger will *automatically* offer to let you take advantage of this feature, under two circumstances:

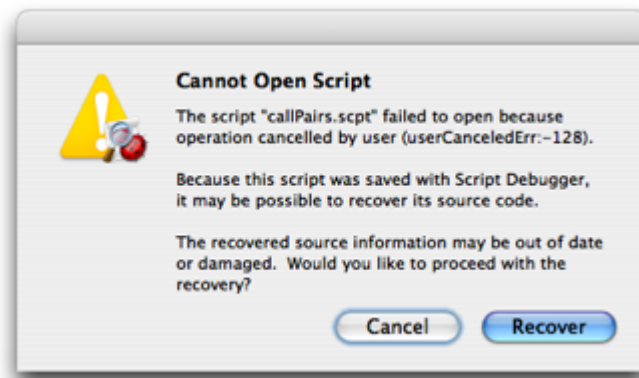
- If you attempt to open a compiled script that **targets an application which [must be launched](#)** in order for AppleScript to [decompile](#) it. For example, suppose BBEdit is not running and you open a compiled script that targets BBEdit. BBEdit has a dynamic dictionary, so AppleScript wants to launch it in order to display the script. Script Debugger detects this and can [optionally](#) intervene, presenting this dialog:



You can proceed to open the script (and allow BBEdit to launch) if you wish, but perhaps the overhead of launching an application just to read a script seems unwarranted. If this script was saved with Script Debugger, it contains a text version, and you can click Open As Text to open that instead. Thus you can read the script without launching BBEdit. (But to *compile* the script you will have to let AppleScript launch BBEdit.)

If the script does *not* contain a text version, the Open As Text button will not be present.

- If you attempt to open a compiled script that **targets a missing application**. In this case, on Mac OS X 10.4 (Tiger) and earlier, after you cancel out of the AppleScript “Where is...” dialog, Script Debugger will offer you a chance to open the script’s text version, if it has one:



(This won’t happen on Mac OS X 10.5 and later because, there, when you cancel out of the “Where is...” dialog, AppleScript opens the script anyway, displaying [raw Apple event codes](#) if necessary.)



To **save a script** with Script Debugger:

- Choose File > Save.
- Choose File > Save As. This creates a new file or, if you wish, overwrites the original file.
- Choose File > Save A Copy As. This creates a new file but the script window continues to show the original file.

Scripts can be saved in [various formats](#). If you are creating a new file, options for specifying the desired format appear in the Save dialog. Alternatively, you can **specify format options** by choosing from these hierarchical menus:

- File > Script Format
- File > Application Options (if it's an [applet](#))

To save a script, the script must be [compiled](#). If you wish to save a script without compiling you can [save it as text](#).

A compiled script (or application) can also be [exported as a run-only script](#).

Further Details:

[Formats](#)
[Run-Only Script](#)
[What Is Saved](#)
[File Owner](#)
[Spotlight and Quick Look](#)

Formats

Script Debugger can save scripts in three basic forms: as a [compiled script file](#), as an [applet](#) (application), or as [text](#).

In each case, you have various options about the resulting format and other details.

A compiled script (or application) can also be [exported as a run-only script](#).

Further Details:

[Compiled Script](#)
[Application](#)
[Text](#)

[Run-Only Script](#) 



Compiled Script



Script Debugger can save [compiled script files](#) in three formats. You can choose a format either from the File > Script Format hierarchical menu or from the Save dialog.

- **Compiled Script (Data Fork).** A file with the compiled [bytecode](#) in the [data fork](#). This is the default format created by the current version of Apple's Script Editor. It is backwards compatible to all versions of Mac OS X and to very late versions of AppleScript in Mac OS 9.
- **Compiled Script (Bundle).** A [bundle](#) (package) with the [bytecode](#) as a [data fork](#) compiled script file inside the bundle. This format was introduced in Panther (Mac OS X 10.3) and is not backwards compatible to earlier systems. It has the advantage that you can store ancillary files inside the bundle, but be warned that some applications do not understand this format.
- **Compiled Script (Resource Fork).** A file with the compiled [bytecode](#) in the [resource fork](#). This is the oldest format and is compatible with all Macintosh systems and all versions of AppleScript.

Keeping the bytecode in the resource fork is, however, also the riskiest format. You can accidentally strip away a compiled script's [resource fork](#). You might, for example, save the compiled script file on a non-HFS filing system (true Unix, or Windows), or send it through email without compressing it, or open it with some badly behaved script editor application.

Warning: A compiled script saved in [debug mode](#) will not run normally in other environments (and will not even open in Apple's Script Editor). Unless that's what you intend, be sure to save the script in normal mode when you're finished debugging it.



Application



A [compiled](#) script can be saved as an application, traditionally known as an *applet*. An applet is a stand-alone application. When opened in the Finder, the script runs. The applet's script can be edited in Script Debugger by opening it with File > Open or by dropping the applet onto Script Debugger's Dock or Finder icon. You can save a script as an applet and leave the script open in Script Debugger. This allows you to test the script from the Finder and then easily edit it in Script Debugger.

Script Debugger has some further features for helping you test a script that is destined to be saved as an applet. You can test individual [handlers](#) in the applet, and you can debug the applet [while it is running](#).

Script Debugger can save applications in two formats. You can choose a format either from the File > Script Format hierarchical menu or from the Save dialog.

- **Application (Carbon).** This format is compatible with Mac OS X and with late versions of earlier systems that use CarbonLib.
- **Application (Bundle).** This format was introduced in Panther (Mac OS X 10.3) and is not backwards compatible to earlier systems. It has the advantage that you can store ancillary files inside the [bundle](#).

There also used to be a "Classic applet" format — an applet compatible with systems and AppleScript versions *before* Mac OS X, which cannot be run as Mac OS X-native. Mac OS X 10.5 (Leopard) has abandoned Classic, so such an applet cannot run at all. Script Debugger 4.5 will not save in this format. It can *open* a Classic applet, but when you try to save you will be compelled to choose another format. If you need to save a Classic applet, use an earlier version of Script Debugger.

Besides the format, you can also set further options for the behavior of the resulting application. To do so, use the checkboxes in the Save dialog, or choose from the hierarchical File > Application Options menu. (You can also add Stay Open and Show Startup buttons to the script window's [toolbar](#). They change their names and icons to indicate whether the setting is on or off.)

- **Show Startup Screen.** The script's [description](#) is used as a "splash screen" when the applet starts up. This splash screen also contains buttons allowing the user to quit or run the applet.
- **Stay Open.** An applet that does not stay open runs its script when opened and then automatically quits. An applet that does stay open does not automatically quit after running its script (the user can choose its Quit menu item to quit it later). This is useful if, for example, the applet runs a handler periodically at idle time.

Warning: An application saved in [debug mode](#) will not run normally (when launched, it will initiate an [external debugging](#) session in Script Debugger). Unless that's what you intend, be sure to save the application in normal mode when you're finished debugging it.

 [Compiled Script](#)

[Text](#) 



You can **save a script as text**, without compiling. The result is an ordinary text file (in UTF-8 encoding, with an initial BOM).

To do so, do either of the following:

- Choose File > Save, File > Save As, or File > Save A Copy As, and choose Text from the popup menu in the Save dialog.
- Choose File > Script Format > Text.

You may need to save a script without compiling for a variety of reasons:

- You want to save your work, but the script doesn't compile (and you don't have time to figure out why right now).
- You want to store the script in a form that's guaranteed to be readable on another computer. (A [compiled script file](#) might fail to open on another computer for a variety of reasons.)

If you wish, you can **specify line endings** for a text file. To do so, use the Text Line Endings popup menu in the File > Save As dialog. In general you should not have to do this. If you leave the line endings setting at As Is (Mixed), line endings will be left alone. Any other setting will force line endings to be set at some specific value, and this can alter the functionality of your script (as explained [here](#)). (To view line endings as they are now in your script, you can [show invisibles](#).)



Run-Only Script



A run-only script contains the script's compiled [bytecode](#) but does not contain the further information needed to decompile and display it. A run-only script is typically used as a way of distributing a script so that other users can run the script but cannot view or modify the script's source code.

To **save a script as run-only**:

- Choose File > Export > Run-Only Script. (You can also add an Export Run-Only button to the script window's [toolbar](#).)

The resulting Save dialog contains the same [format options](#) as for an ordinary [compiled script](#) or [application](#).

It also contains a checkbox, "Make Bundled Scripts Run-Only". This is useful in cases where you've added extra scripts to a [script bundle](#) or [application bundle](#). If you don't check it, the bundle's main script will be saved as run-only, but the extra scripts in the bundle will not be.

A run-only script cannot be read or edited ever again, even by you, its creator! This is why Script Debugger implements this feature as a form of export. After exporting as run-only, your original script is unaffected (and therefore remains editable). If you edit your original script and you wish to propagate the changes to the run-only version of the script, export it again.





What Is Saved



Script Debugger saves the following information into a compiled script file:

- The compiled script [bytecode](#).
- The script's [description](#), if any.
- [Persistent information](#) such as the current values of script properties.
- [Script window](#) state (such as its size and position) and [view settings](#).
- The open or closed state of the [result drawer](#) — but this will be restored the next time you open the file only if you have checked the “Remember Result drawer state” [General preference](#).
- Apple Event Log window tab [view and mode settings](#).
- [Libraries](#).
- [Expressions](#).
- [Breakpoints](#), if the script is saved in [debug mode](#).



File Owner



When a file is opened from the Finder, it is opened by the application that owns it. Therefore, since applications other than Script Debugger (such as Apple's Script Editor) can claim ownership of script files, you might want to **control the ownership** of files created by Script Debugger.

To do so, go to the [General preference pane](#) and select from the Saving radio buttons.

- If you choose **Script Debugger is always creator**, any file saved by Script Debugger will have its creator code changed to Script Debugger.
- If you choose **Keep original creator**, creator codes of saved files not created by Script Debugger will be left untouched, but files created by Script Debugger will have Script Debugger's creator code.
- If you choose **No creator**, any file saved by Script Debugger will have no creator code.

In theory, Mac OS X decides a file's owner based on the internally stored creator code, if there is one; otherwise, it uses the filename extension (*.scpt* for a compiled script file, *.scptd* for a compiled script bundle, *.applescript* for a script text file). In reality, the interplay between these two modes of determining a file's owner is somewhat unpredictable. Script Debugger offers the option to save files with no creator, thus *forcing* Mac OS X to fall back on the filename extension to determine ownership. Additionally, you can check **Default editor for OSA scripts, applets and droplets** in the [General preference pane](#) as a quick way of setting Script Debugger as the owner for files with the relevant filename extensions.



Spotlight and Quick Look



Script Debugger supports both Spotlight and Quick Look access to saved compiled script files.

Spotlight

Spotlight is an indexing technology, introduced in Mac OS X 10.4 (Tiger). It keeps track of files and their contents and allows you to search rapidly for a file based on its name or its contents. If you can remember a word or two used in your file, you can find it quickly, rather than having to remember what folder it's in.

Script Debugger contains a "Spotlight importer" for compiled script files. This means that if Script Debugger is present on your computer, AppleScript compiled scripts are searchable with Spotlight.

Besides the script's name and contents, you can search in its [description](#), and you can search on the name of the [OSA language](#) that the script uses.

If Spotlight stops finding the contents of compiled scripts on your machine, it is likely that the Spotlight indexing system has become confused. You can compel your primary hard disk to rebuild its Spotlight index by saying `sudo mdutil -E /` in the Terminal; for more information, see the [mdutil](#) documentation.

Quick Look

Quick Look is a technology, introduced in Mac OS X 10.5 (Leopard), for viewing a preview of the contents of a file without the overhead of opening that file in the application that [owns](#) it.

Script Debugger contains a "Quick Look generator" for [compiled script files](#) and [applets](#). This means that the system is provided with the information it needs to translate your file into a preview that Quick Look can present. If a script has been saved with Script Debugger, it will be viewable with Quick Look "in color" (that is, with all the AppleScript compiled script [text formatting](#)); otherwise, it will appear in Quick Look as plain text.

Note that because you're just "peeking" at the script's text with Quick Look, there is none of the overhead involved with actually opening the script: there is no [decompilation](#) and therefore there is no need to [launch](#) any targeted applications.



Description

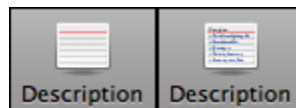
A script can have a *description*. This can serve as a reminder to the developer, an explanation to users, and as a splash screen in an [applet](#).

To **access a script's description**:

- Choose File > Description.

The description appears in a dialog. It consists of styled text. Text styling will be maintained in an applet's splash screen.

Optionally, Script Debugger can show you whether a script has a description attached. Choose View > Customize Toolbar and drag the Description icon into the [toolbar](#). The icon indicates whether the description has any text. The illustration below shows the icon for a script without and with a description, respectively.





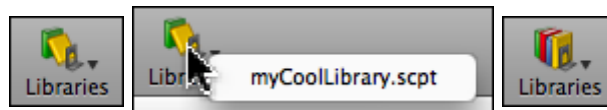
Library



Script Debugger allows parts of a script to be kept in a separate, secondary compiled script file. The secondary file is called a *library*. With a library, your scripts can easily make use of common code. If the code in the library changes, scripts that use the library inherit the changed code automatically when they are compiled.


If you're going to make extensive use of libraries, it might be a good idea to add the Libraries icon to your [script window toolbar](#). To do so, choose View > Customize Toolbar and drag the Libraries icon into the toolbar. The icon indicates whether the script has libraries. In addition, it contains a popup menu which you can use to add a library from your Script Libraries folder *instantly* to your script.

The illustration below shows: (1) the toolbar icon for a script without libraries; (2) the use of the toolbar popup menu to add a library to a script; (3) the toolbar icon for a script with libraries.



The popup menu automatically looks for library files in *Application Support/Script Debugger 4.5/Script Libraries*, both in your user *~/Library/* and in the top-level */Library/* directory. Therefore you are encouraged to keep library files in one of these locations, so that the popup menu can find them and you can take advantage of this feature. (The popup menu also looks for a *Script Libraries* folder in the same folder as Script Debugger itself, but use of this feature is not encouraged.)

To **add a library file** to your script:

- If the library file is in a Script Libraries folder (in one of the specified locations), use the toolbar icon popup menu as shown above.
- Or, choose File > Libraries (or click the Libraries toolbar button) to summon the Libraries dialog, and then:
 - If the library file is in your Script Libraries folder, click the first button () to access the popup menu and choose a library file. (Option-choose to reveal the library file in the Finder.) This popup menu is the same as the toolbar icon popup menu.
 - Or, click the second button (+) to choose any compiled script file.
 - Or, drag a compiled script file from the Finder into the dialog.

The Relative To popup determines how you want Script Debugger to locate this library file in future:

- **Absolute** means that the library file is sought in its current location.
- **Document** means that the library file is sought in the same location relative to the current script. The current script must have been saved or this option won't be enabled. For example, you might start with the library file in the same folder as the

current script. Both files can then be moved, and as long as they are moved together (so that they remain in the same folder), the library file will be found.

- **Application Support** means that the library is sought in the same location relative to your Script Libraries folder. This is naturally the default when you use the shortcuts described above for adding a library from your Script Libraries folder.

The way a file is listed to show its path in the Libraries dialog changes depending on your choice in the Relative To popup. (If there isn't enough room to see a full path, you can widen the dialog.)

To **examine or edit a library file**:

- Choose File > Libraries and:
 - Double-click the library file listing to open the file for editing.
 - Option-double-click the library file listing to reveal the file in the Finder.

A script that uses Script Debugger's library feature will *run* in other contexts — the library resources are invisibly merged into the script when the script is saved, in a way that AppleScript understands — but it cannot be *edited* except by Script Debugger. In order to distribute to others a script which uses libraries, in a form that can be edited with any script editor application, you will want to *flatten* the script. This means that the contents of all library files on which the script depends are visibly incorporated into the contents of the script itself.

To **flatten a script**:

- Choose File > Export > Flattened Script.

The Script Debugger Libraries mechanism may remind you of the AppleScript `load script` command, but it has several advantages over `load script`:

- There is no need for your code to load anything, as Script Debugger does the loading for you.
- With `load script`, the loaded material becomes a script object within your script, whereas with Script Debugger's library feature, the loaded material is blended with your script.
- With `load script`, you have to specify or calculate a pathname in code, whereas Script Debugger helps keep track of libraries for you.

Here is a further [technical discussion](#) about how the Libraries feature works.

Further Details:

[Technical Details About Libraries](#)



Technical Details About Libraries



Here are some further technical details about how Script Debugger's [Libraries](#) feature works.

How Libraries Are Attached to a Script

A library file (that is, a script that appears in your script's Libraries dialog) is *not* dynamically loaded every time you run your script. It is reloaded when Script Debugger [compiles](#) the script (and when you open the Libraries dialog). These are the only times when Script Debugger cares about where the library file is located on disk (a location that it derives in accordance with your settings in the Relative To popup menu in the Libraries dialog).

The important implication here is this: The mere fact that you change the contents of a library file does *not* mean that your compiled script that uses the library file will automatically acquire the change. Rather, it is up to you to *compile* your script that uses the library, as a way of causing any libraries it uses to be reloaded. Luckily, compilation happens any time you alter your script and then run it; or, if your script hasn't been altered and doesn't need compilation, you can **force it to be compiled** (and therefore to reload its library files) by choosing Script > Recompile or by pressing the Recompile button in the script's toolbar (hold down the Option key to make it replace the Compile button).

Once loaded, a library is (invisibly) present as part of your compiled script. The resulting compiled script file can be executed in *any* script runner environment. (But to *edit* it in another script editing environment without flattening it first will probably cause the invisibly present library to be stripped out.)

Duplicate Library Definitions

Unlike AppleScript's `load script` command, top-level entities of a library file effectively become top-level entities of your script. In theory, this means that their names can clash with other top-level entities. For example, if a library file contains a top-level handler `howdy ()`, and if your script already has (or you eventually give it) a handler (or any other top-level entity) named `howdy`, you've got two top-level things named `howdy`, and that's illegal.

Script Debugger helps you with this situation. If you try to compile a script where there's a name conflict between libraries or between a library and the main script, Script Debugger puts up a dialog warning you of the problem ("The following duplicate library definitions have been found"). You won't be able to compile the script under these circumstances, unless you click Continue in this dialog (in which case Script Debugger steps out of the way and lets AppleScript deal with the duplication).

Such a name conflict includes the existence of two run handlers. In other words, given that you have a library file (or more than one library file) plus the main script, only one of those may contain top-level executable code, because such code constitutes an implicit run handler, and you can't have two run handlers in a script.

An Obscure Top-Level Entity Bug

There's a bug in AppleScript that can cause a top-level entity in your main script to be confused with a top-level entity in a library. For example, suppose your library file goes like this:

```
property x : "hello"
on greet()
    display dialog x
end greet
```

And suppose your main script goes like this:

```
property y : "goodbye"
greet()
```

When you then execute your script, the resulting dialog says "goodbye", not "hello", even though `greet` refers to `x` ("hello") and no code anywhere refers to `y` ("goodbye"). The reason is that AppleScript has confused `x` with `y`.

A safe approach is this: if a script is to be used as a library file, wrap the whole script up in a script object, like this:

```
script myLibrary
    property x : "hello"
    on greet()
        display dialog x
    end greet
end script
```

That way, when the library is loaded, what's loaded is a single object — the script object. The script object keeps the namespace clean, preventing name confusions. (In the example above, the way to refer to `x` in your script would be as `myLibrary's x`, and the way to call `greet()` in your script would be to say `tell myLibrary to greet().`)

Another solution is not to have a property declaration and a handler definition in the same library file, thus avoiding the situation that triggers the bug.



Manifest



It is the nature of AppleScript that if a scriptable application or scripting addition on which a [compiled script](#) depends is missing, the script might not run, or could [refuse to open](#) for reading and editing.

To help with this problem, Script Debugger can generate a *manifest* for your script - a list of the scriptable applications and scripting additions on which it depends. That way, if you intend to move the script to another machine or send it to another user, you'll be forewarned about the script's dependencies.

To **see a script's dependencies**:

- Choose File > Manifest. (You can also add a Manifest button to the script window's [toolbar](#).)

The script's dependencies are listed by category (**Application Dependencies** and **Scripting Addition Dependencies**), along with commands that the script directs at each one. For example, this script:

```
tell application "Finder" to beep
```

would list the Finder in the Application Dependencies category, and would list StandardAdditions.osax in the Scripting Addition Dependencies category along with the beep command. You can reveal a required application or scripting addition in the Finder. You can also view its [dictionary](#). If you select a command before pressing the Dictionary button, the dictionary displays the information for that command.

A special situation can arise when a script depends on a scripting addition which is now missing. The scripting addition's terminology can't be resolved during decompilation, so raw Apple event codes appear, and AppleScript has no way to tell you what the trouble is or what scripting addition is missing (because, unlike applications, scripting additions are not explicitly targeted by name). To help with this problem, the manifest also lists Apple events in your script that can't be resolved (**Unknown Apple Events**). You can select such an Apple event in the list and Script Debugger will search for the Apple event code in the online database at www.osaxen.com in an attempt to identify the missing scripting addition.

(Another solution to this problem, if the script was last saved with Script Debugger, might be to [open the script as text](#); this will show the English-like terminology for the missing command, which might help you work out what scripting addition it comes from.)

The dialog also lets you **export and save as a text file** the information it presents, in either XML (**Save XML Report**) or plain text format (**Save Text Report**).

Script Debugger cannot detect coercions provided by scripting additions. (For example, [Jon's Commands](#) can coerce a script object to text. If your script performs such a coercion and you try to run it in the absence of Jon's Commands, the script will break, and Script Debugger won't be able to help you figure out why.) That's because *nothing* can detect them. They do not involve any terminology and are not documented in a scripting addition's dictionary. This is an AppleScript flaw.

 [Library](#)



Explore



The biggest challenge for the AppleScript programmer is figuring out **what to say to a scriptable application**. Script Debugger gives you powerful tools for exploring a scriptable application so that you can quickly write successful scripts targeting it.

- You can explore an application's [dictionary](#). Script Debugger helps you [navigate](#) and [search](#) the dictionary, and lets you [view](#) dictionary information fully and clearly.
- You can explore a scriptable application's [objects in real time](#). While the application is running, you can see the names and values of the attributes (elements and properties) that it actually has at that moment — and if those attributes are objects, you can see *their* attributes, and so forth. You can copy a reference to an attribute into your script. You can even change a property value, without writing a script.

Further Details:

[Dictionary Explorer](#)

Dictionary

One of Script Debugger's most helpful features is its **display of the [dictionary](#)** of a scriptable application.

Script Debugger makes it very easy for you:

- To [open](#) the dictionary of a scriptable application or scripting addition.
- To [navigate](#) the dictionary.
- To [view](#) the information in the dictionary in a variety of ways.

Further Details:

[Open Dictionary](#)
[Dictionary Window](#)
[Dictionary Views](#)





Open Dictionary



You can open an application's [dictionary](#) in several different ways, depending on what you're doing and what's convenient.

- There are various **basic ways** to open [any application or scripting addition's dictionary](#).
- If an **application is running**, you can [open its dictionary directly](#).
- If you're editing a **script that targets an application**, you can [open that application's dictionary directly](#).
- If you've **already worked with** an application, Script Debugger remembers this fact, and you can [open the dictionary from a list of previously used applications](#).
- If the dictionary you want to open is a **special dictionary** — it's an installed scripting addition, or it's the AppleScript Studio dictionary, or it's the dictionary of AppleScript itself — you can [open it directly](#).

You can **open multiple windows on the same dictionary**. Bring a dictionary window to the front, and choose Dictionary > Open in New Window. This makes it easy to view different pieces of information simultaneously, or to have both a dictionary and an [explorer](#) open for the same application.

Note: In some cases, opening an application's dictionary will require the application to be running. If this is the case, and the application is not running, then when you ask to open the application's dictionary, Script Debugger will [launch the application](#), which may cause a delay. However, once Script Debugger has opened an application's dictionary, it [caches](#) the dictionary. Thus, having opened such an application's dictionary, you can now close the application and its dictionary and later open its dictionary again and this time Script Debugger will *not* have to launch the application.

Further Details:

[Open Any Dictionary](#)
[Current Applications](#)
[Current Context](#)
[Known Applications](#)
[Scripting Additions](#)





Open Any Dictionary



If an application is not [running](#) and you've never [worked with it before](#), and if you are not [currently targeting it](#) in a script, you can **open its dictionary** in the following ways:

- Choose File > Open. This brings up a dialog where you can choose anything. If what you choose is an application or scripting addition, its dictionary will open.
- Choose File > Open Dictionary > Application. This brings up a dialog where you can choose an application or scripting addition to open its dictionary.
- Locate the application or scripting addition in the Finder and drag its icon onto Script Debugger's icon (possibly onto Script Debugger's icon in the Dock).
- Locate the application or scripting addition in the Finder and drag its icon onto a [Script Debugger script window](#). Script Debugger will put up a dialog asking what you want to do. One of the options is to open the application's dictionary.



Current Applications



If an application is **running right now**, you can **quickly open its dictionary**.

- Choose File > Open Dictionary. In the resulting hierarchical menu, you'll see a list headed "Running Scriptable Applications". Choose an application from the menu, and its dictionary will open.
- Alternatively, if what you want to open is the dictionary of the **frontmost application**, use Script Debugger's Dock menu. This contains an item, Open XXX Dictionary, where "XXX" is the frontmost application. Choose it, and Script Debugger will come to the front with that dictionary open.



Current Context



If you're editing a script that contains **a tell block targeting an application**, you can **quickly open that application's dictionary**.

- Select anywhere *inside the tell block* targeting the application whose dictionary you want to open. (This is to put your selection into the desired [tell context](#).) Then choose File > Open XXX Dictionary, where "XXX" will be the name of the application. Alternatively, control-click to choose the same menu item from the contextual menu.



Known Applications



When you ask Script Debugger to work with an application, Script Debugger remembers this fact and puts the application in a [known applications list](#). You can **open a known application's dictionary quickly**:

- Choose File > Open Dictionary. In the hierarchical menu, you'll see a set of menu items entitled "Known Applications". Choose an application to open its dictionary.
- Choose Window > Inspectors > Known Applications if necessary, to show the [Known Applications inspector](#). In the inspector window, double-click an application's listing (or select it and click the Dictionary button) to open its dictionary.



Scripting Additions



Certain **special dictionaries** can be opened directly in Script Debugger.

- To open the dictionary of an **installed scripting addition**, choose File > Open Dictionary > Scripting Additions. (The keyboard shortcut for this command is likely to become one of your most frequently used shortcuts.)
- To open the **AppleScript Studio dictionary**, choose File > Open Dictionary > AppleScript Studio.
- To open **AppleScript's internal dictionary**, choose File > Open Dictionary > AppleScript.

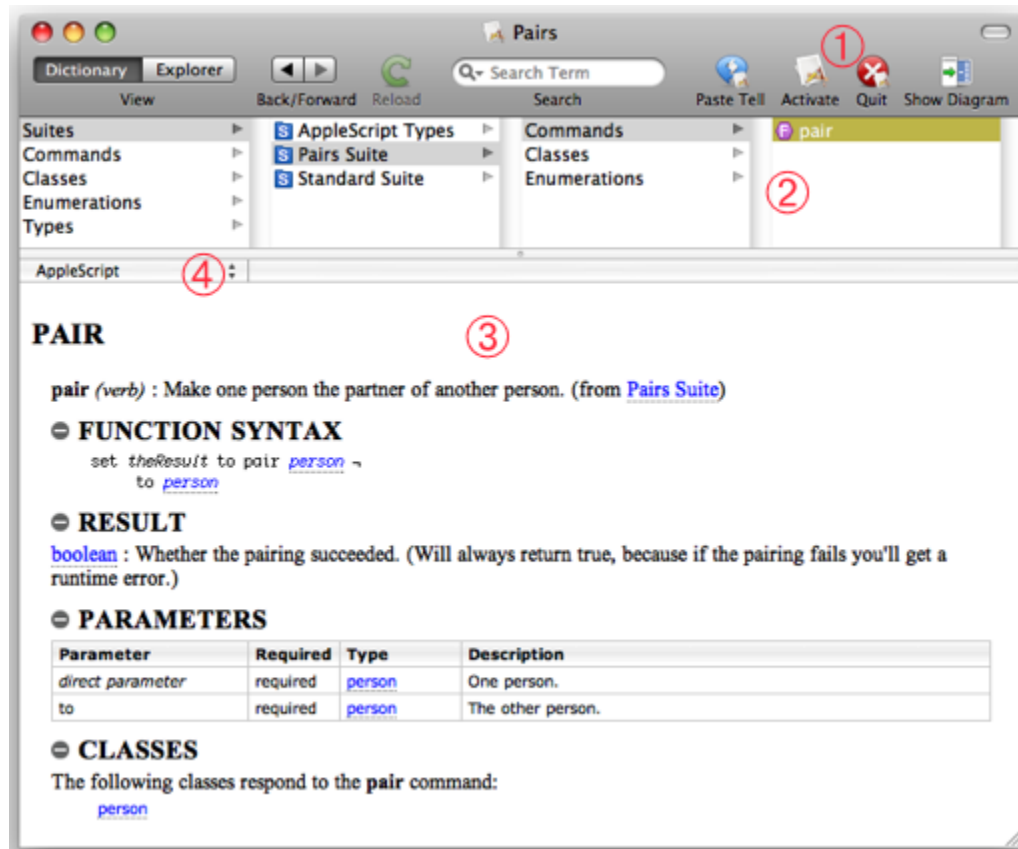
AppleScript's internal dictionary is the 'aeut' resource, contained in the AppleScript component file. It can be helpful for explaining certain [terminology clashes](#), as it is the only way to learn what terms are defined by AppleScript. (For example, why can't you name a global variable "keystroke"? This dictionary tells you.) Within this dictionary, only the AppleScript Suite, the Standard Suite, the Text Suite, and the Type Names Suite are actually enabled. Even though you can see the other suites, they are turned off and do not actively define any terminology (and should be ignored).



[Known Applications](#)

Dictionary Window

Meet Script Debugger's dictionary display!



(The dictionary illustrated here is for a tiny scriptable application called Pairs, especially written for test purposes. You don't have this application, so don't go looking for it on your hard disk!)

Here are the parts of the dictionary window:

1. The [toolbar](#).
2. The [browser](#). Select in the browser to see the corresponding dictionary entry in the info pane. (The browser is replaced with search results when you [search](#) the dictionary.)
3. The [info pane](#). It displays an entry or entries from the dictionary. To determine what entry is displayed here, select in the [browser](#) (or in the results of a [search](#)).
4. The [Format popup menu](#).

Further Details:

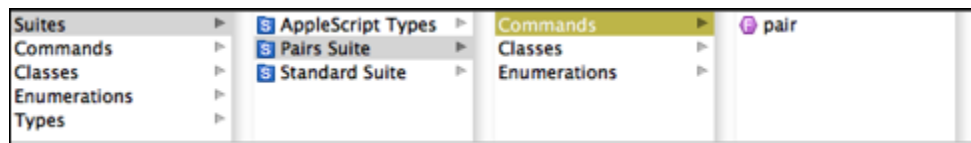
[Browser](#)
[Dictionary Info Pane](#)
[Hierarchies and Diagrams](#)
[Search in Dictionary](#)
[Look Up Definition](#)
[Back and Forward](#)
[Miscellaneous Dictionary Actions](#)

 [Open Dictionary](#)

[Dictionary Views](#) 



The browser at the top of the [dictionary window](#) is the primary way of navigating the dictionary.



Select in the browser to display the corresponding dictionary entry in the [info pane](#).

In the above illustration, we have clicked the Suites entry in the first column so as to pick a suite in the second column. In the second column, we have clicked the Pairs Suite so as to pick a category in the third column. In the third column is the actual selection, the Commands listing. This means that all commands in this suite are listed in the info pane. (It also means that all commands in this suite are listed in the fourth column. It so happens that this tiny suite has only one command.)

To select more than one listing in a column, Shift-click to select a range of entries, or Command-click to select multiple individual entries. Choose Edit > Select All to select all entries in a column, thus displaying information for all of them in the info pane.

So, in the above illustration, you could Command-click Enumerations in the third column. Both Commands and Enumerations in the third column would be selected, and so all commands and enumerations in the Pairs Suite would be displayed in the info pane.

You can also use arrow keys to navigate the browser, though it is probably more common to use the mouse.

Here is a discussion of the various [rows, columns, and icons](#) you'll see displayed in the browser (and in [search results](#)).

Further Details:

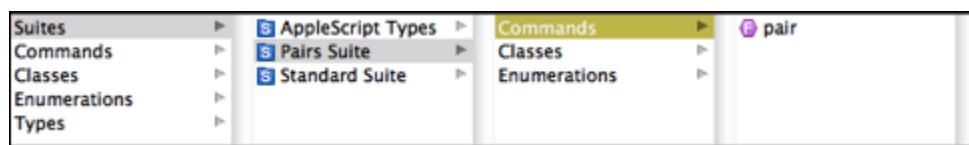
[Types of Entities Shown in the Dictionary Browser](#)



Types of Entities Shown in the Dictionary Browser



This is a list of the various types of entities shown in the [browser](#) at the top of the dictionary display, as well as in results from searching the dictionary (in the [dictionary window](#) or through the [Look Up Definition](#) inspector).



Suite

A suite (**S**) is an artificial grouping of entities created by the developer of the application's scripting model. A suite can contain any other type of entity (except another suite), and in most dictionaries, every dictionary entry is part of some suite. At the same time, in Script Debugger's display of the dictionary, all dictionary entries are also accessible, starting in the first column, *without* using the Suites entry.

So, in the above illustration, we could have started with the Commands entry in the first column to see the `pair` command listed in the second column and in the [info pane](#).

Command

A command is a verb, something that you tell an application or one of its objects to do. The icon distinguishes between a function (**F**), which returns a result, and a plain command (**G**), which does not. Command parameters are marked in a search result by **P**.

Event


An event (**E**) is a message sent by an application to your code. Your code can receive this message through an event handler. For example, Folder Actions are implemented through events. If you want something to happen when files are added to a folder, you must implement a handler called `adding folder items to`, so that System Events can call it by sending the corresponding event to your script.

Dictionaries do not always distinguish events from commands, so Script Debugger will sometimes report events as commands. For example, most of the "commands" listed in the AppleScript Studio dictionary are really events, but Script Debugger has no way to know this.


Class

A class (**C**) is a datatype. A class can have attributes — properties (**P**) and elements. Most of what you do in scripting an application involves working with properties and elements. You get and set the values of properties, and you manipulate elements in various ways (asking for particular elements or lists of elements, creating new elements, deleting elements, and so forth).



Record

A record () is a sort of lightweight class, typically used only in communicating data between an application and your script. Script Debugger distinguishes records from classes only when the target application has an [sdef-based dictionary](#) (because only an sdef-based dictionary can draw such a distinction).

Type





A type () is a built-in datatype, such as `boolean` or `string`. It is not always obvious what the distinction is between a Type and a Class. In some contexts, the difference is that a Class can have attributes (properties and elements), but some dictionaries fail to draw this distinction consistently.














Enumeration




An enumeration () is a datatype whose value is always one of a predefined list of constants (). For example, the `saving` parameter of the `close` command is either the constant `yes` or the constant `no` or the constant `ask`. An enumeration's constants are called its *enumerators*.

In the dictionary [info pane](#), each enumeration is listed, with its enumerators, on a page of its own. A value type is shown as a hyperlink with the enumeration name. Click this link to see the enumeration's own page, listing its enumerator values. For example, the `saving` parameter of the `close` command is listed as the `saving` enumeration. Click that to see the `saving` enumeration on its own page, where the enumerators `yes`, `no`, and `ask` are listed.

Scripting Addition

This category appears only in the Scripting Additions dictionary display. Script Debugger collects the dictionaries of all installed scripting additions into a single dictionary, so this category lets you browse an individual scripting addition. Individual scripting additions are marked as to their location, namely the ScriptingAdditions folder in the Library at the system () , computer () , user () , or network () level.

Summary of Symbols		
	# in a folder	Enumeration
	# in a circle	Constant (Enumerator)
	C in a circle	Class
	C in a hexagon	Command
	E in a hexagon	Event
	F in a hexagon	Function (Command with result)
	P in a square	Parameter
	Pr in a square	Property
	R in a circle	Record
	S in a folder	Suite
	T in a circle	Type
	! in a triangle	PowerPC-only*
	Mac OS X-style X	System**

	iMac display	Computer**
	Person silhouette	User**
	Network globe	Network**

* The PowerPC-only icon appears when Script Debugger is running natively on an Intel-based machine and a scripting addition is PowerPC only. It alerts you to the fact that Intel-native AppleScript environments, including Script Debugger, will not load this scripting addition. (But a Carbon applet, or any application that executes AppleScript scripts and is not a universal binary and therefore runs under Rosetta, *will* load such a scripting addition.)

** Scripting Addition icons reflect the location of the *Library/ScriptingAdditions* folder containing the marked scripting addition file: */System*, top level, the user's home folder, or the network, respectively.



The info pane of the [dictionary window](#) displays the dictionary entry. Script Debugger's display of dictionary entries has these useful features:

- **Completeness.** Script Debugger displays *all* the information in a dictionary, where other script editor applications may omit some information.
- **Collapsibility.** Each section of the dictionary info display is collapsible. For example, in the first illustration below, if you click on the "minus" symbol (▢) to the left of the **RESULT** heading, the contents of the Result area are hidden. In a dictionary with large sections, this can be a considerable space saver.
- **Hyperlinking.** Everything in blue **with a dotted underline** in a Script Debugger dictionary is a link. You click this link to follow it. For example, in the first illustration below, every time the `person` class is mentioned, it's a link which you can click to jump to the dictionary's entry on the `person` class. If you're in doubt about what any linked word means, click on it! (There's no penalty for doing so, as you can always [come back](#) afterwards.)

Note: links with *solid* underlines are external (i.e. somewhere on the Internet) and will be displayed in your web browser.

- **Cross-Referencing.** Script Debugger analyzes the dictionary, draws some conclusions, and displays the resulting information. For example, at the bottom of the first illustration below, Script Debugger tells you what classes can be the object of the `pair` command. In the second illustration below, showing the `alias` entry from BBEdit's dictionary, Script Debugger tells you every command that takes an alias as a parameter, and every class that has a property that's an alias. Every cross-reference is a hyperlink.
- **Extra Information.** Script Debugger provides extra information about built-in AppleScript types. For example, in the second illustration below, `alias` is a built-in AppleScript type. The **Description** section is extra information, coming not from the dictionary of the application (BBEdit) but from Script Debugger itself. This extra information is instructional and can be especially helpful to AppleScript beginners.

PAIR

pair (*verb*) : Make one person the partner of another person. (from [Pairs Suite](#))

FUNCTION SYNTAX

```
set theResult to pair person ~  
to person
```

RESULT

[boolean](#) : Whether the pairing succeeded. (Will always return true, because if the pairing fails you'll get a runtime error.)

PARAMETERS

Parameter	Required	Type	Description
direct parameter	required	person	One person.
to	required	person	The other person.

CLASSES

The following classes respond to the **pair** command:

[person](#)

The above illustration shows the typical features of a command as displayed in Script Debugger. The command's syntax is demonstrated by a template ("Function Syntax"), which you can [insert into your script](#). The result and parameters are clearly shown. (Not illustrated above: optional command parameters are displayed in grey.) Classes ([person](#)) and types ([boolean](#)) are hyperlinks. Classes that can be the object of this command are cross-referenced and hyperlinked.

ALIAS

alias (*noun*), *pl aliases* : An alias to a file or a folder on disk.

DESCRIPTION

An *alias* object is very much like a [file](#) object. You can form an alias specifier in just the same way as you form a file specifier, and an alias object can often be used in the same places where a file object would be used. But there are some important differences:

- If an alias specifier uses a literal pathname string, then the item on disk that it represents must exist at compile time. (But starting with AppleScript 2.0, introduced in Mac OS X 10.5 "Leopard," this is no longer the case.)
- If an alias specifier uses a string variable, then the item on disk that it represents must exist when the specifier is encountered at runtime.
- An alias value can be assigned directly to a variable.
- An alias can continue pointing to an item on disk even if the item is moved or renamed.

PROPERTIES

Property	Access	Type	Description
POSIX path	get	string	The POSIX path of the file.

WHERE USED

The **alias** class is used in the following ways:

- direct parameter to the [insert clipping](#) command/event
- file** property of the [project item](#) class/record
- file** property of the [window](#) class/record
- file** property of the [document](#) class/record
- saving in** parameter of the [close](#) command/event
- to** parameter of the [save](#) command/event
- to** parameter of the [export](#) command/event
- url** property of the [project item](#) class/record

The above illustration shows the typical features of a class as displayed in Script Debugger. The class's properties are clearly listed. Types (`file`, `string`) are hyperlinks. Commands where this class is a parameter, and classes where this class is a property or element, are cross-referenced and hyperlinked. This particular class is a built-in AppleScript type, so extra information about the type is provided in the Description section.

The [Find dialog](#) works in the info pane, and can be a useful way to reach desired information quickly.

◀ [Browser](#)

[Hierarchies and Diagrams](#) ▶



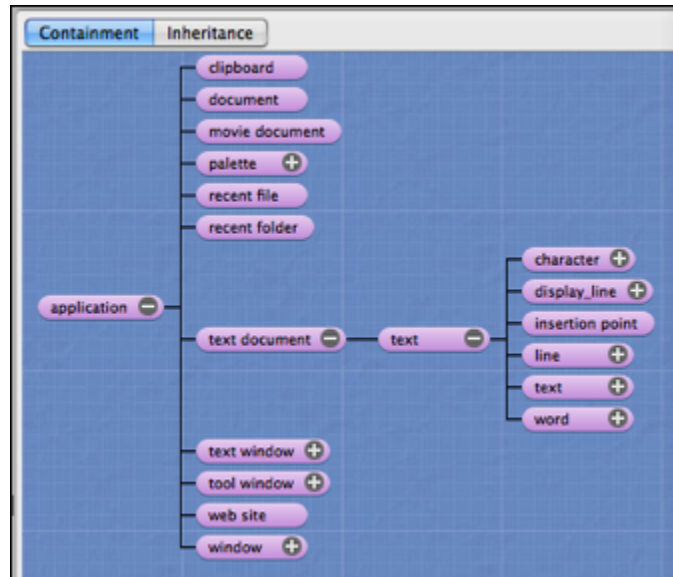
Hierarchies and Diagrams



Classes in a dictionary can be understood as arranged in two **hierarchies**. To see them, choose Dictionary > Show Diagram (or click Show diagram in the [dictionary window's](#) toolbar). The button at the top of the diagram drawer lets you choose between hierarchies — the **containment hierarchy** or the **inheritance hierarchy**.

- The **containment** hierarchy reflects the fact that an object has attributes (properties and elements), and an attribute can be another object. Thus, in theory, it should be possible to start at the “top” of the hierarchy (which is usually the single instance of the application class) and describe the relationships between classes as a tree. This tree is sometimes referred to as the application’s [object model](#). The containment hierarchy expresses a “has-a” relationship among classes.
- The **inheritance** hierarchy is an artifice originally introduced as a way of making dictionaries shorter. For example, in the Finder, `folder` and `disk` are two different classes, but they have many properties and elements in common. For instance, they both have an `entire contents` property saying what’s in them, and they can both have `folder` elements and `file` elements reflecting the hierarchy of items on disk. Thus it saves space, and makes conceptual sense as well, to separate out the `entire contents` property and the `folder` elements and `file` elements, along with all the other attributes shared by folders and disks, and express them as a separate class (here called `container`). The `folder` class and the `disk` class are then said to *inherit* from the `container` class, so that they share these properties and elements by virtue of this inheritance. The inheritance hierarchy expresses an “is-a” relationship among classes.

Script Debugger’s dictionary display can flatten the display of inherited attributes in the [info pane](#). [Click here](#) to read more about this feature.



The above illustration (showing BBEdit's containment hierarchy) is typical of what you'll see in the diagram drawer. You can do three things here:

- Click any class's name in the diagram to **see the information for that class** displayed in the [info pane](#). Thus, the diagram drawer is an additional way to navigate the dictionary.
- Click the **+** or **-** button at the right end of any class's name, to **expand or collapse** the hierarchy shown in the diagram from that point.
- Choose from the popup menu at the bottom of the drawer, to **change the root** of the diagram. Here, the `application` class is the root of the diagram, but you can change this to any class that has attributes. This is convenient as a way of "hoisting" part of the diagram, and is also valuable in the case of defective dictionaries, where the object model is faulty and fails to form a single coherent hierarchy.

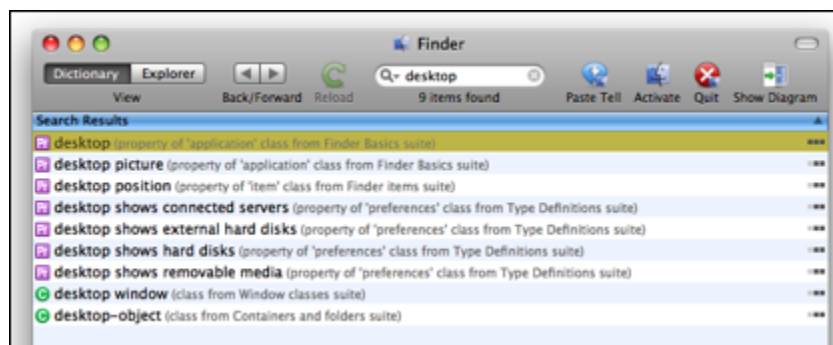


To **search a dictionary**:

- In the dictionary window, show the [toolbar](#) (if it isn't showing already) and **use the Search field**. Type a term to search for in the Search field, and press Return to initiate the search.

The popup menu at the left end of the Search field (the icon looks like a magnifying glass) determines **what categories of entity** to search in, as well as **what parts** of each entity's info to search in. (For example, you might find it useful to disable searching in a Description, since this can lead to false positives.) The popup menu also remembers recent searches.

You are doing a literal search. If the text you type is found anywhere within the material being searched, you have a hit. For example, if your search includes entity names, searching on "lect" will find a class that has an attribute called "selection".



Search results are displayed in place of the [browser](#) at the top of the [dictionary window](#). Results are grouped alphabetically into relevance rankings. (Above is shown the result of searching for "desktop" in the Finder's dictionary.) Click a listing to display it in the [info pane](#). Thus, searching is another way to navigate the dictionary.

[You might wish to see a list of the [types of entity you can find here](#), and the meanings of the icons.]

To **remove search results** and restore the browser at the top of the dictionary window:

- Click the X icon at the right end of the Search field (or click in the Search field and press the Esc key).

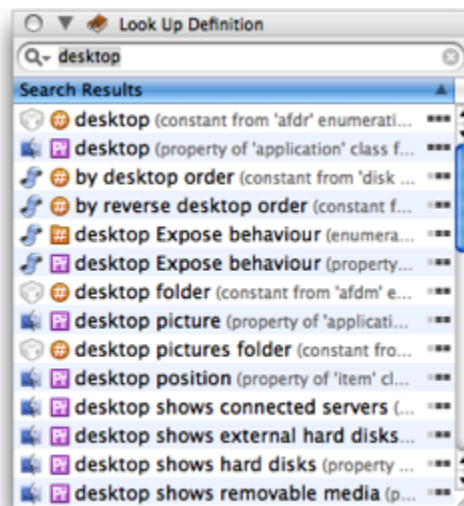
Another way to search the dictionary is through the [Look Up Definition](#) feature.



Look Up Definition



You can [search a dictionary](#) without being *in* that dictionary, either from within the script you're working on, or from an inspector window.



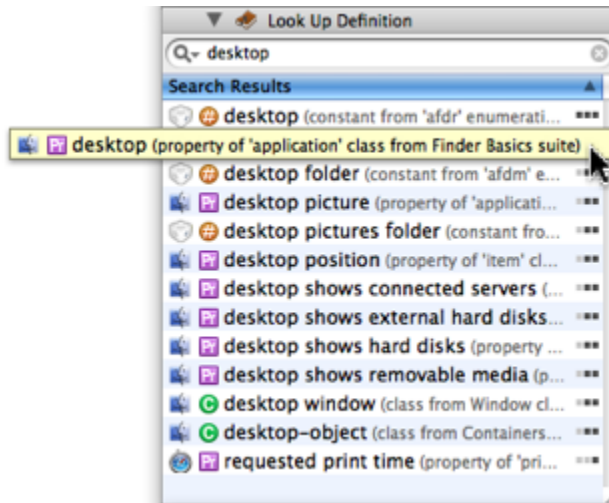
- To search the dictionary from the [Look Up Definition inspector](#), first make sure the inspector is showing (if not, choose Window > Inspectors > Look Up Definition). The search field here is just like the [Search field](#) in the dictionary window toolbar. In particular, it has the same popup menu at the left end, with the same options determining what aspects of the dictionary to search. It also has five additional options (at the top of the menu) determining *what* dictionary to search:
 - You can choose whether to include [scripting additions](#) in your search.
 - You can choose whether to include [AppleScript's own internal dictionary](#) in the search. (This option is good for helping to track down [terminology clashes](#).)
 - You can choose whether to search the [tell context](#) application. This is the application being targeted at the selection point in the frontmost script window.
 - You can choose whether to search scriptable running applications.
 - You can choose whether to search all applications in the [known applications list](#).

Searching the known applications list can be time-consuming the first time that you do it, because dictionaries must be loaded and applications may have to be [launched](#). But it will be fast after that because the dictionary is cached. When you start to search known applications, a dialog will appear warning you of any applications that must be launched in order to load their dictionaries, giving you a chance to prevent any or all of these applications from launching (in which case their dictionaries will not be searched or cached).

- To search a dictionary from within a [script window](#), select a term in the script's text, and choose Search > Look Up Definition. (Alternatively, choose Look Up Definition from the script window's contextual menu.) This enters the selection into the Look Up Definition inspector and performs the search. The settings already present in the Look Up Definition inspector search field popup menu will apply to this search.

[You might wish to see a list of the [types of entity you can find here](#), and the meanings of the icons.]

The Look Up Definition inspector cannot be widened, so the entire content of a result usually cannot be displayed. To see the full content of a result, hover the mouse over the result; a tooltip will appear, showing the full result.



Having performed a search in the Look Up Definition inspector, and having obtained results, to **display a result in its dictionary**:

- Double-click the result.



Back and Forward



The dictionary display remembers everything displayed in the [info pane](#). It's as if each display in the info pane were a web page. As in a web browser, you can **go back to previously viewed "pages"**, and then forward again. To do so:

- Choose Dictionary > Back or Dictionary > Forward (or use the convenient keyboard shortcuts). Alternatively, use the Back/Forward buttons in the dictionary window's toolbar.



Miscellaneous Dictionary Actions



This page discusses some miscellaneous actions that you can perform in a dictionary window.

- **Quit.** If the application whose dictionary you are looking at is running, you can make it quit. Use the Quit button in the dictionary window toolbar, or choose Dictionary > Quit XXX, where “XXX” is the name of the application.

If you do this in the Finder’s dictionary, you’ll get a warning dialog, since quitting the Finder is not something one usually wants to do. If you *do* quit the Finder, you can relaunch it by clicking the Launch button.

- **Launch or Activate.** You can start up the application whose dictionary you are looking at, or, if it’s already running, you can bring it to the front. Use the Launch or Activate button in the dictionary window toolbar (they are the same button), or choose Dictionary > Launch XXX or Dictionary > Activate XXX (they are the same menu item), where “XXX” is the name of the application.
- **Paste Tell.** You can insert a tell block into your script, targeting the application whose dictionary you are currently looking at. Use the Paste Tell button in the dictionary window toolbar, or choose Dictionary > Paste Tell. If you hold down the Option key, or if no script window is open, a new script window is created and the tell block is inserted. Otherwise, the tell block is inserted at the current insertion point or wrapping the selection in the frontmost script window. Your existing code will not be overwritten.

If what you’re looking at in the dictionary is a *command*, a template for giving that command will be inserted as well. If what you’re looking at in the dictionary is an *event*, a template for an event handler for receiving that event will be inserted.

There are other ways to issue a [Paste Tell command](#).

If you alter the contents of your ScriptingAdditions folder while you have the Scripting Additions dictionary window open — or if you’re a developer writing a scriptable application, and you alter that application’s dictionary while you have the application’s dictionary window open — a Caution icon (⚠) will appear in the dictionary window title bar. Click it, and you’ll get a dialog prompting you to reload the application’s dictionary.



◀ [Back and Forward](#)

Dictionary Views

Several menu items allow you to change aspects of the [information display](#) in a dictionary.

- You can change the [size](#) of the text.
- You can set whether a class's information should include properties and elements [inherited from its superclass](#).
- You can set whether to display [extra documentation](#) supplied by dictionary authors, as well as Late Night Software's own explanations of AppleScript's built-in classes.
- You can set whether to reveal the [raw Apple event codes](#) for dictionary terms.
- You can reveal the [raw XML](#) underlying Script Debugger's display of the dictionary information.

These settings apply to the current dictionary window, not to all dictionary windows, and they are persistent (that is, they are remembered the next time you open the same dictionary window).

Further Details:

[Size](#)
[Inheritance](#)
[Extra Documentation](#)
[Apple Event Codes](#)
[Dictionary Format](#)



You can **change the size of the text** in the dictionary display. To do so:

- Choose Dictionary > Larger Text or Dictionary > Smaller Text, possibly several times until you get a suitable text size. (You can also add a Text Size button to the dictionary window's [toolbar](#).)



Inheritance



Script Debugger's dictionary display can flatten the display of [inherited attributes](#) in the [info pane](#). So, for example, looking at info for the Finder's `folder` class shows you the info for the container class (because `folder` inherits from `container`) and the `item` class as well (because `container` inherits from `item`).

To **flatten the display of inherited attributes**:

- Choose the Dictionary > Show Inherited Properties and Dictionary > Show Inherited Elements menu items. If these menu items are checked, the display of inherited attributes is being flattened.

There is a different sort of dictionary flattening, which Script Debugger performs automatically. In some dictionaries, an entry is repeated multiple times. For example, the information about the `application` class might be distributed over two entries, each in a different suite. (So, for example, in TextEdit's dictionary, some `application` class information is in the Standard Suite, some of it is in the TextEdit Suite.) In this case, Script Debugger will display the entry in its multiple locations, but in both places it combines the information from both entries. Thus, no matter which instance of the entry you look at, you will see *all* the information about that entry.



Extra Documentation



The [sdef dictionary](#) format (introduced in Tiger, Mac OS X 10.4) allows entries in the dictionary to be more extensive than previously. Besides a mere comment, dictionary authors can now include multiple formatted paragraphs of explanation.

Technically, such information appears in <documentation> tags in the XML that constitutes the sdef dictionary.

You can **toggle the display of extended explanatory material**. To do so:

- Choose Dictionary > Show Extra Documentation. If the menu item is checked, the extended explanatory material is being displayed.

Script Debugger itself also includes supplementary dictionary information that is “injected” into the dictionary display, such as the Description of the `alias` datatype shown here. This supplementary information is instructional and can be especially helpful to AppleScript beginners.

ALIAS

alias (*noun*), *pl aliases* : An alias to a file or a folder on disk.

DESCRIPTION

An *alias* object is very much like a [file](#) object. You can form an alias specifier in just the same way as you form a file specifier, and an alias object can often be used in the same places where a file object would be used. But there are some important differences:

- If an alias specifier uses a literal pathname string, then the item on disk that it represents must exist at compile time. (But starting with AppleScript 2.0, introduced in Mac OS X 10.5 “Leopard,” this is no longer the case.)
- If an alias specifier uses a string variable, then the item on disk that it represents must exist when the specifier is encountered at runtime.
- An alias value can be assigned directly to a variable.
- An alias can continue pointing to an item on disk even if the item is moved or renamed.

PROPERTIES

Property	Access	Type	Description
POSIX path	get	string	The POSIX path of the file.

WHERE USED

The **alias** class is used in the following ways:

- direct parameter to the [insert clipping](#) command/event
- file property of the [project item](#) class/record
- file property of the [window](#) class/record
- file property of the [document](#) class/record
- saving in parameter of the [close](#) command/event
- to parameter of the [save](#) command/event
- to parameter of the [export](#) command/event
- url property of the [project item](#) class/record

 [Inheritance](#)

[Apple Event Codes](#) 



Apple Event Codes



From AppleScript's point of view, the [purpose of the dictionary](#) is to translate between the English-like terminology seen by you, the human programmer, and the raw codes used to construct Apple events. The dictionary display can show you these raw Apple event codes. This can be useful when you want to analyze a raw Apple event (as displayed in the [Apple Event Log window](#), for instance), or when you need to track down a [terminology clash](#).

To **toggle the visibility of raw Apple event codes**:

- Choose View > Show Raw (Chevron) Syntax. If the menu item is checked, Apple event codes are showing.

You can also see raw Apple event codes in [scripts](#) and the [Apple Event Log window](#).



Dictionary Format



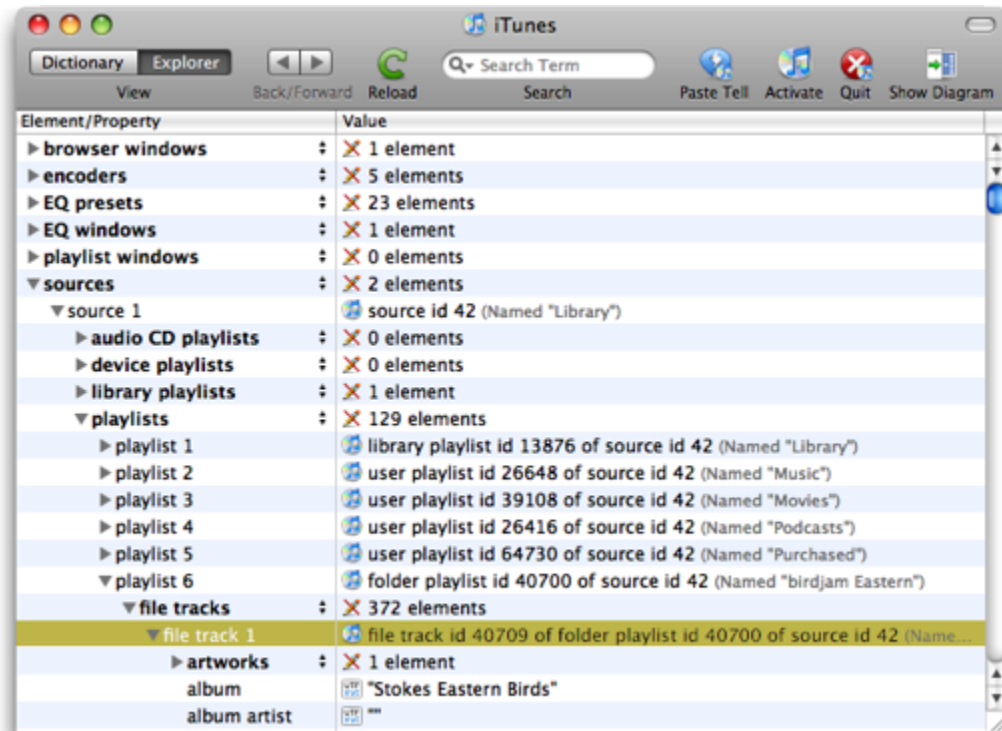
The Format popup menu above the [info pane](#) of the [dictionary window](#) changes the form in which the information is shown. By default, there are two choices here, “AppleScript” and “[sdef](#) Outline”.

Script Debugger’s dictionary display is generated from some XML created behind the scenes. This XML is in a sense the “real” contents of the dictionary. The AppleScript setting in the Format popup menu transforms this XML into a nice layout for convenient viewing. The other choice, “sdef Outline”, permits you to view the original XML. The XML presentation has syntax coloring and code folding (meaning you can click on a minus-sign or plus-sign at the left of a line to collapse or expand everything hierarchically below that line).

Users are free to add other transformations. Such additional transformations are installed in *~/Application Support/Script Debugger 4.5/Dictionary Stylesheets* and will then appear in the Format popup menu.



Meet Script Debugger's Explorer!



The explorer lets you probe the objects of a running scriptable application, in real time. You can easily discover an application's [object model](#), insert object references into your scripts, and see the values of object properties — and even *change* those values.

To **see a dictionary window's explorer**:

- [Open a dictionary window](#) and choose Dictionary > Explorer View (or click the Explorer button in the toolbar).

You can **open multiple windows on the same dictionary**. Bring a dictionary window to the front, and choose Dictionary > Open in New Window. This makes it easy to view different pieces of information simultaneously, or to have both a dictionary and an explorer open for the same application.

The illustration above shows iTunes's dictionary explorer, as it appears on my machine. Its parts are:

1. The [toolbar](#).

2. The **info pane**. This area is where you see the information about a running application. It is an [explorer view](#). (Explorer views appear in several other places in Script Debugger's interface.)

Read on, to learn about:

- What's shown in an [explorer view](#).
- [Things you can do](#) in an explorer view.
- How to open part of an explorer view as a [separate window](#).

Note: A more focussed way to probe an application's objects is the [Tell Context inspector](#). It watches where you are working in a script window and probes the attributes of the [current tell target](#).

Further Details:

[Explorer View](#)
[Explorer Details](#)



Explorer views appear in many places in Script Debugger's interface. For example:

- The [Explorer pane](#) of a dictionary window.
- The [Tell Context inspector](#).
- The [variables pane](#) and [expressions pane](#) of a script window's [result drawer](#).
- [Best view](#) in a [viewer pane or window](#), such as:
 - The [result pane](#) of a script window's result drawer (or separate result window).
 - An [error dialog](#).
 - A separate [viewer window generated](#) from any explorer view.

This page describes **what is shown** in in an explorer view. A [further page](#) discusses **what you can do** in an explorer view.

An explorer view is an outline, a hierarchy. To **expand or contract** the hierarchy beneath a line:

- Click that line's disclosure triangle (or select that line and press the Right Arrow or Left Arrow key).

Element/Property	Value
▶ browser windows	✕ 1 element
▶ encoders	✕ 5 elements
▶ EQ presets	✕ 23 elements
▶ EQ windows	✕ 1 element
▶ playlist windows	✕ 0 elements
▼ sources	✕ 2 elements
▼ source 1	🌐 source id 42 (Named "Library")
▶ audio CD playlists	✕ 0 elements
▶ device playlists	✕ 0 elements
▶ library playlists	✕ 1 element
▼ playlists	✕ 129 elements
▶ playlist 1	🌐 library playlist id 13876 of source id 42 (Named "Library")
▶ playlist 2	🌐 user playlist id 26648 of source id 42 (Named "Music")
▶ playlist 3	🌐 user playlist id 39108 of source id 42 (Named "Movies")
▶ playlist 4	🌐 user playlist id 26416 of source id 42 (Named "Podcasts")
▶ playlist 5	🌐 user playlist id 64730 of source id 42 (Named "Purchased")
▼ playlist 6	🌐 folder playlist id 40700 of source id 42 (Named "birdjam Eastern")
▼ file tracks	✕ 372 elements
▼ file track 1	🌐 file track id 40709 of folder playlist id 40700 of source id 42 (Name...
▶ artworks	✕ 1 element
album	🎵 "Stokes Eastern Birds"
album artist	🎤 "

Here are the kinds of things shown in an explorer view:

- **Element collections** are shown in bold with plural names and disclosure triangles. The “value” displays the element count. Expand an element collection to see the individual elements in the collection. Each individual element’s “value” is a reference to that element (along with information about its name, if available). In the illustration above (from the iTunes explorer), the user has opened the application’s sources collection, revealing source 1 (whose name is “Library”), and from there has drilled on down through the element hierarchy to reveal file track 1 of playlist 6 of source 1. The sources, playlists, and file tracks entries are all element collections.
- **Objects** are shown with disclosure triangles as well. The “value” is a reference to the object, as returned by the application, and is badged with the application’s icon (to indicate that it’s an object belonging to that application). Expand the object reference to see the object’s elements and properties. Above, the user has opened file track 1 of playlist 6 of source 1, revealing its artworks elements collection, its album property, and so forth.
- **Lists and Records** are shown with disclosure triangles too. The “value” displays the item count (but a [Dictionary preference](#) can change this). Expand the entry to see the individual items of the list or record. Below is a picture showing an example (from the Finder’s explorer).





▼ selection	list of 2 items
▶ item 1	document file “motrip2.rtf” of folder “Desktop” of folde...
▶ item 2	folder “PoppeaVolume” of folder “Desktop” of folder “m...

- **Script Objects** are shown with disclosure triangles too. (A common place to see this is in the [variables pane](#).) Expand a script object’s listing to see its script properties and its top-level script objects. The illustration below is from a script containing a property x and a script object ss, which itself has a property x and a script property sss, which also has a property x.

▶ parent	«script AppleScript»
▶ AppleScript	«script AppleScript»
▼ ss	«script ss»
▼ sss	«script sss»
x	3
x	2
x	1

- **Other datatypes** are shown as individual lines without triangles. The value is shown as AppleScript would display it. So, in the above illustration of iTunes’s explorer, the album property of this object (file track 1 of playlist 6 of source 1) is a string, “Stokes Eastern Birds”.

Actually, the album property is Unicode text, and Script Debugger lets you know this with the “UTF XVI” badge that appears next to it. Text badges you may see are:

	Unicode text (UTF-16)	[Common]
	UTF-8	[Rare]
	Styled text	[Rare]
	International text	[Rare]

Also, if a value is a valid file reference, it is badged with the icon of the corresponding item on disk.

Read on to learn [what else you can do](#) in an explorer, and how to generate [separate viewer windows](#) so you can focus on the details of particular values.

[Explorer Details](#) 



Explorer Details



Here are details about things you can do in an [explorer view](#).

Element/Property	Value
▶ browser windows	✖ 1 element
▶ encoders	✖ 5 elements
▶ EQ presets	✖ 23 elements
▶ EQ windows	✖ 1 element
▶ playlist windows	✖ 0 elements
▼ sources	✖ 2 elements
▼ source 1	🌐 source id 42 (Named "Library")
▶ audio CD playlists	✖ 0 elements
▶ device playlists	✖ 0 elements
▶ library playlists	✖ 1 element
▼ playlists	✖ 129 elements
▶ playlist 1	🌐 library playlist id 13876 of source id 42 (Named "Library")
▶ playlist 2	🌐 user playlist id 26648 of source id 42 (Named "Music")
▶ playlist 3	🌐 user playlist id 39108 of source id 42 (Named "Movies")
▶ playlist 4	🌐 user playlist id 26416 of source id 42 (Named "Podcasts")
▶ playlist 5	🌐 user playlist id 64730 of source id 42 (Named "Purchased")
▼ playlist 6	🌐 folder playlist id 40700 of source id 42 (Named "birdjam Eastern")
▼ file tracks	✖ 372 elements
▼ file track 1	🌐 file track id 40709 of folder playlist id 40700 of source id 42 (Name: ...)
▶ artworks	✖ 1 element
album	🌐 "Stokes Eastern Birds"
album artist	🌐 ""

- **Move a listing into a script.** Individual lines of an explorer can be copied or dragged into a script. (Drag from the first column.) In a dictionary's explorer, an [alternative](#) is to choose Dictionary > Paste Tell (or use the Paste Tell button in the toolbar). If what you copy into your script in this way is an object reference, it will be conveniently wrapped in a tell block if necessary.

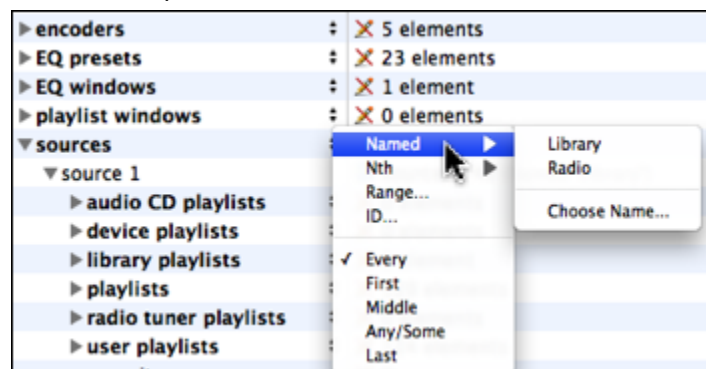
You can also copy from the Value column. For example, instead of "album of file track 1 of playlist 6 of source 1", you might like to paste "Stokes Eastern Birds" into your script. To do so, use the contextual menu and choose Copy Value, or hold Shift and choose Edit > Copy. Then paste wherever you want the value to go.

- **Change a value.** A value is writeable, and can be changed, unless it is badged with the crossed-out pencil icon (✎) indicating that it is not writeable. Select the desired line and press Return (or Enter), or choose Edit Value from the contextual menu. The entry in the Value column will become editable, and you can change it. When you're done, press Enter to set your change, or press Esc to cancel and leave the value untouched. Be careful! If what you are exploring is an object belonging to an external application, *you are changing the actual value of an actual property of an actual object in the actual application!* In the illustration above, the album property is an example.
- **Change an enumerated value.** If a value has little up-and-down arrows (⌵⌶) to its left, it's an editable enumeration. The arrows are a popup menu, and you can click

them to get the menu and change the value (or see what the other possibilities are). In the illustration below, the shuffle and song repeat properties are examples.

persistent ID	✗ "BBB9D920D054748A"
shuffle	⚙ false
size	✗ 7.4724730592E+10
song repeat	⚙ off
special kind	✗ none
time	✗ "44:19:56:11"
visible	✗ false

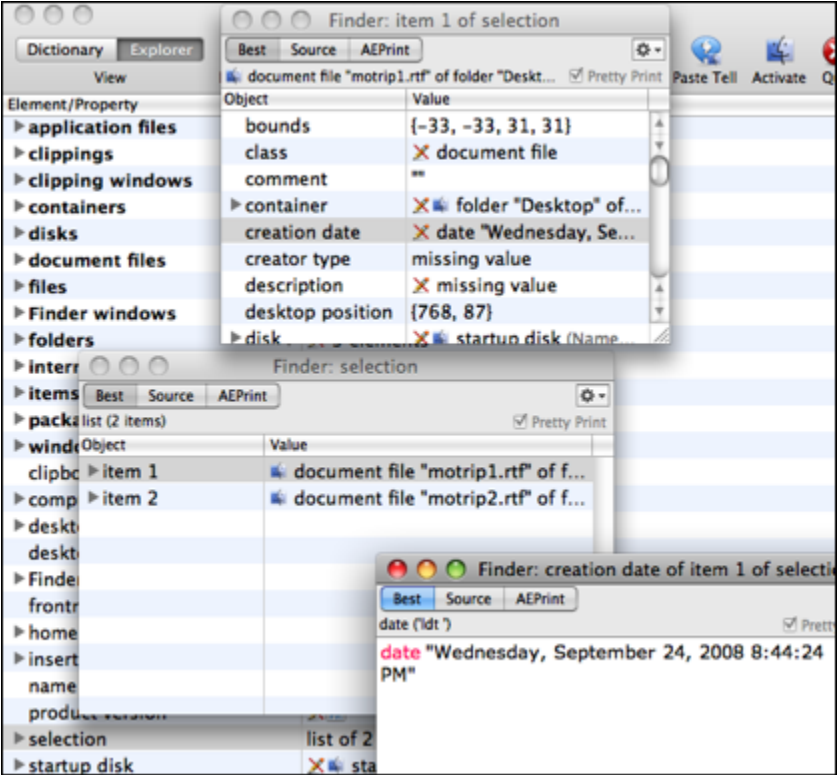
- **Change an element specifier.** Element collections have little up-and-down arrows (⚙) at the right end of the left column. These allow you to change the element specifier used to gather the collection. By default, we get every whatever-it-is (for example, the list of sources was gathered by asking for every source). With the popup menu, you can get a different range or an individual element. The screen shot below shows how you can change from every source to the particular source named "Library".



- **Get a class definition.** To learn more about a class (as opposed to an object), use the contextual menu and choose Show Definition. For example, if I control-click on the sources line, in the illustration above, and choose Show Definition, the window switches to Dictionary view with the source class selected.
- **Refresh the explorer.** An explorer is not live all the time. Information is gathered when you initially open the explorer and when you expand a triangle. If the situation in the target application changes, the explorer view information will need refreshing. To refresh it, select a line and choose Dictionary > Reload (or choose Reload from a line's contextual menu). That line, and everything exposed that's hierarchically deeper, will be gathered afresh. (To refresh the *whole* window, hold down the Option key as you choose Dictionary > Reload. But be careful, since Script Debugger may have to ask the target application for a lot of information.)
- **Generate a separate viewer window.** To focus in on a particular entity, double-click it. (Alternatively, select the entity and control-click it, and in the contextual menu, choose Open Viewer.) This creates a new [viewer window](#) showing the entity's value.

You might wish to generate a [separate viewer window](#) simply because it is a better way to view information that you are particularly interested in. For instance, if it is an object, Best View in the viewer window *is itself an explorer view*, so you can see that object by itself without the other contents of the main explorer — and then you might generate yet another separate viewer window focusing even more tightly. If a value is a native datatype, the display in the viewer window may be easier to read (for example, in the case of a long

string). The illustration below shows a cascade of viewer windows. From the Finder explorer, we separate off the selection in its own window; from that window, we separate off item 1 of selection in its own window; from that window, we separate off creation date of item 1 of selection in its own window.



Another reason for opening a separate viewer window is that sometimes it shows the information better than the original explorer view does. For example, in the Finder's explorer, the selection object is a list. You can expand this list's entry to see its items, as shown in the illustration below:

selection	list of 2 items
item 1	document file "motrip2.rtf" of folder "Desktop" of folde...
item 2	folder "PoppeaVolume" of folder "Desktop" of folder "m...

...but if you expand one of those items in the explorer, you do *not* see the elements and properties of that object. Instead, you see a lot of error messages. The reason is that we're asking the Finder for things like bounds of item 1 of selection, and the Finder chokes on this sort of expression. The solution is to double-click the selection line, opening a separate viewer window. Here, the items and their elements and properties are fully displayed. The reason is that we've already fetched the selection, so in this window we're asking the Finder for bounds of item 1 of (get selection), which works.

In general, don't be perturbed by error messages in an explorer. These are marked in orange, with a stop-sign icon, as in the illustration below (CSS palette, from BBEdit's explorer), and they indicate that Script Debugger asked for an element or property and the application responded, in good order, with a runtime error.

container	empty
CSS palette	Get failed: the requested data type cannot be supplied (errAECantSupplyType:-10009)

An error message, or a `missing` value result, are valid responses from the application. An «empty» value (as in `container` in the above illustration) is supplied by Script Debugger to indicate that the application has failed to return any value at all. This is useful information because if you assign such a non-value to a variable in a script, that variable will be undefined (try telling BBEdit to `set x to container` and then displaying `x` and you'll see what this means).



[Explorer View](#)



Script Debugger gives you powerful, unique tools to help you develop your AppleScript code. From a short one-off script to a complicated program, you'll easily and quickly create code that works.

You can think of the AppleScript development cycle in three stages:

[Edit](#) your code.

[Enter](#) code, [view](#) it, [navigate](#) it, [search](#) it, [modify](#) it.

[Run](#) your code.

[Compile](#) code, [execute](#) it, [time](#) it, get the [result](#), and [log](#) Apple event communication between your code and scriptable applications.

[Debug](#) your code.

[Step](#) through code, set [breakpoints](#), watch [values](#) change as your code progresses. Understand what your code does, line by line!

Further Details:

[Edit](#)
[Run](#)
[Debug](#)



Edit



Script Debugger is a dedicated editor for AppleScript programs. It's designed to make it as easy as possible for you to edit your AppleScript code.

- Get to know the [script window](#).
- Learn about [editing and navigating](#) in the script window.

Further Details:

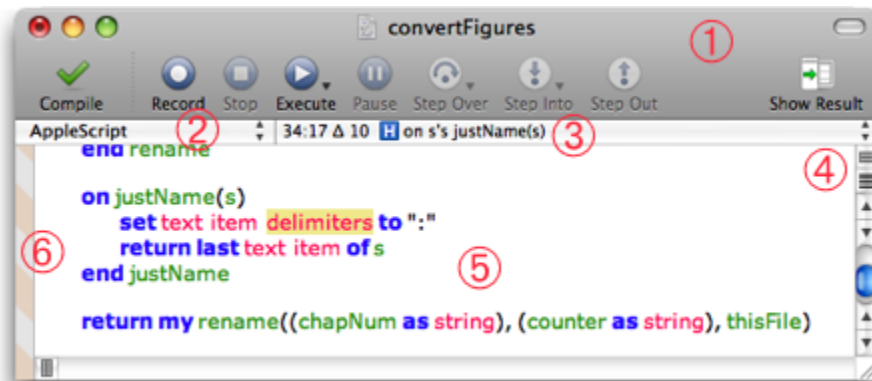
[Script Window](#)
[Editing and Navigation](#)



Script Window



Meet Script Debugger's script window environment! This is where you will read, edit, run, and debug your scripts.



1. [Toolbar](#)
2. [Language](#) popup
3. [Navigation](#) bar and "table of contents" popup
4. [Split](#) and [wrap](#) icons
5. Text area. This is where you [edit](#) the script. You can alter your [view](#) of the script's contents.
6. Gutter (the column at the left edge of the window). Information appears here, such as [error marks](#), [line numbers](#), and (when debugging) [breakpoints](#) and [code coverage marks](#).

The script window appearance is configurable, so you might want to save a certain size, shape, position, and appearance as the [default](#), so that whenever you create a new script, it has the look that you prefer.

Information about opening and saving scripts is [here](#).

Further Details:

[Toolbar](#)
[View](#)
[Language](#)
[Default Script Size and State](#)





The toolbar at the top of a window contains controls relevant to the functionality of that window. So:

- The toolbar at the top of a [script window](#) contains controls for running and debugging your script.
- The toolbar at the top of a [dictionary window](#) contains controls for switching between dictionary view and explorer view, for reloading the selected entry, for activating or quitting the target application, and so on.
- The toolbar at the top of the [Apple Event Log window](#) contains controls for determining what is logged and in what format, and can contain controls for running the corresponding script.

In general, the functionality in a toolbar is available also from menus, so feel free to hide the toolbar if you want to conserve screen space. However, some toolbar controls are not available elsewhere. For example, the [Search field](#) in the dictionary window toolbar has no exact parallel elsewhere in the interface.

To **show or hide** a toolbar:

- Choose View > Show Toolbar or View > Hide Toolbar (it's the same menu item). Alternatively, click the "lozenge" (oblong) button at the right end of the title bar.

To **customize** a toolbar:

- Choose View > Customize Toolbar. This brings up the Customize Toolbar dialog, where you can determine what controls appear in the toolbar, as well as setting the icon style and size. Alternatively, control-click in the toolbar icon area to bring up the contextual menu that lets you do the same things. Even without the Customize Toolbar dialog, you can Command-drag an icon in a toolbar to change its position or remove it from the toolbar. Your changes apply (immediately) to the toolbars of all windows of the same type. Thus, customizing the toolbar of a script window customizes the toolbars of all script windows, and customizing the toolbar of a dictionary window customizes the toolbars of all dictionary windows.



Script Debugger provides many options for helping you view [script window](#) contents.

Besides choosing whether to [wrap](#) long lines, you can also show or hide [invisible](#) characters and [spaces](#), see [line numbers](#) and [tab stops](#), and even view your script's [raw Apple event codes](#).

Many of these options apply in other contexts as well, such as [viewers](#).

Further Details:



[Line Wrapping](#)
[Invisibles](#)
[Spaces](#)
[Line Numbers](#)
[Tab Stops](#)
[Raw \(Chevron\) Syntax](#)



Line Wrapping



Script Debugger lets you choose between wrapping long lines automatically and letting long lines extend to the right. To **toggle line wrapping**:

- Click the wrap icon (at the top of the vertical scroll bar), or choose View > Wrap Lines. If the menu item is checked, line wrapping is turned on (). If not, line wrapping is turned off () , and long lines will extend to the right and a horizontal scroll bar will appear if necessary.

Each setting has its merits. AppleScript is a line-based language, so when you're viewing AppleScript code it can be nice to let long lines extend to the right, so that "a line is a line is a line". On the other hand, scrolling horizontally to see long lines is troublesome, and wrapping solves the problem nicely.



Invisibles

Script Debugger lets you see invisible text characters, such as tabs and return characters. To **see invisible characters**:

- Choose View > Show Invisibles. If the menu item is checked, invisibles are showing.

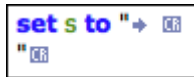
This is particularly useful for viewing string literals. AppleScript has an annoying habit of turning whitespace literals into literal whitespace, which conceals what they are. For example, this script:

```
set s to "\t\r\n"
```

compiles like this:

```
set s to "  
"
```

Thanks a lot, AppleScript! Now you've no idea what that string consists of. But with Script Debugger's Show Invisibles feature, you can instantly find out.



This is particularly useful, as in the above example, with line endings, which can present a complicating factor in AppleScript, as explained [here](#).

Spaces

Script Debugger can show space characters in text. They are shown as small dots. To **show space characters**:

- You must first be showing [invisible characters](#). Now choose View > Show Spaces. If the menu item is checked (and invisibles are showing), spaces are showing.





Line Numbers



Script Debugger can show your [script](#) with line numbers in the margin. To **toggle the visibility of line numbers**:

- Choose View > Show Line Numbers. If the menu item is checked, line numbers are showing.

This can be useful for informational purposes, and for helping you gauge the size of your script. It is also helpful for navigation. The [navigation bar](#) also shows line numbers, and you can [jump](#) to a line by its number.



Tab Stops



Script Debugger can show tab stops in text, the horizontal points at which indentation automatically occurs when a script is [compiled](#) or when a value is [pretty-printed](#). The positions of the tab stops are shown by faint vertical stripes behind your text. To **show tab stops**:

- Choose View > Show Tab Stops. If the menu item is checked, tab stops are showing.



[Line Numbers](#)

[Raw \(Chevron\) Syntax](#)





Raw (Chevron) Syntax



Script Debugger lets you view, in a [script window](#), the raw Apple event codes that constitute a compiled script. This can help explain [terminology clashes](#) and other unexpected AppleScript phenomena, and can give you a deeper understanding of the Apple events that your script is constructing and sending. To **view raw Apple event codes**:

- Choose View > Show Raw (Chevron) Syntax. If the menu item is checked, raw Apple event codes are showing.

You can also see raw Apple event codes in the [dictionary](#) and the [Apple Event Log window](#).



The language popup, at the left above the text area in the [script window](#), determines the OSA ([Open Scripting Architecture](#)) language that the script will use.

Mac OS X provides one OSA language — AppleScript. Script Debugger supplies another OSA language:

- **AppleScript Debugger X.** This is the special version of AppleScript that implements Script Debugger's [debugging](#) features. Whenever debugging mode is [turned on](#), this popup will be set to AppleScript Debugger X.

Additionally, Late Night Software distributes the [JavaScript OSA component](#), which makes JavaScript an OSA language (and gives it the ability to send and receive Apple events).



Default Script Size and State



The appearance of a new empty [script window](#) is up to you. To **set the appearance of new script windows**:

1. Create a new script window and set it up the way you want it — size, [language](#), [view options](#), open or closed state of the [result drawer](#), as well as (if you wish) contents, [description](#), [libraries](#), and [expressions](#). Also, if you like, open the [Apple Event Log](#) and set up its [view and mode settings](#) as you prefer them. Then:
2. Choose Window > Set Default Script Size & State.

A dialog appears, telling you what features your script window has, and giving you a chance to decline to save any of them (except the size and state, which are always saved) as part of the default script window setup.

From now on, new script windows (and their corresponding Apple Event Log tabs) will have these same characteristics.

To open a new script window that **ignores your saved defaults** (without destroying those defaults):

- Hold down the Option key and choose File > New Script (No Defaults).

To revert to Script Debugger's **default appearance** for new windows:

- Choose Window > Reset Default Script Size & State.

Setting the default script appearance causes the model script window to be compiled and saved (in `~/Library/Preferences/Script Debugger Preferences`). Script Debugger opens a copy of this saved file as your new window from then on. (This fact may cause your model window to become “dirty” when you set it as the default, because it was compiled and therefore changed internally even if it contains no code.) Resetting the default script size and state throws away this saved model window.





Editing and Navigation



Script Debugger provides tools to make editing and navigating your script as fast and effortless as possible.

Editing

Script Debugger helps you create, view, and select [block structure](#) and [delimiter pairs](#).

Script Debugger helps you type. It [completes AppleScript terms](#) for you, plus you can define your own [text substitutions](#). And there are other miscellaneous helpful [typing and selection features](#).

You can easily [insert a tell block](#), or any [common control structure](#) or other boilerplate, into your script. You can also [insert content](#) from the Finder or a Dictionary or Explorer window to save typing.

You can [paste a string literal](#), [shift indentation levels](#), [add or remove comments](#), and [interchange tab and space characters](#).

If all of that isn't enough, you can edit your script with an [external editor](#).

Navigation

Script Debugger has powerful [find-and-replace features](#).

You can [split](#) a script window's editing area, so as to view and edit multiple regions of your script simultaneously.

You can [jump](#) to a line by number, or navigate to a section of code within your script by means of the [table of contents menu](#).

Further Details:

- [Block Structure](#)
- [Delimiters](#)
- [Text Completion](#)
- [Text Substitution](#)
- [Miscellaneous Typing and Selection](#)
- [Tell](#)
- [Clippings](#)
- [Inserting Content](#)
- [Shift](#)
- [Comment](#)
- [Tab](#)
- [Splitting the Editor](#)

[Find](#)
[Go To](#)
[Navigate](#)
[External Editor](#)

◀ [Script Window](#)



Block Structure



A pervasive and characteristic feature of AppleScript code is its *block structure* — a `tell` line is balanced by an `end tell` line, a `repeat` line is balanced by an `end repeat` line, and so forth. Script Debugger makes it easy to create, view, and check your code's block structure.

Block Entry (Auto-Closing)

Auto-closing is a text entry feature where, when you type Return in the opening line of a block, Script Debugger automatically creates the closing line of the block for you.

Auto-closing is turned on through an [Editor preference](#):

- **Auto-close AppleScript blocks (end tell, etc.).** If checked, auto-closing is turned on.

To test auto-closing, create a new script and type `repeat`. Now press Return. Script Debugger creates a corresponding end line and positions the insertion point in between, like this:

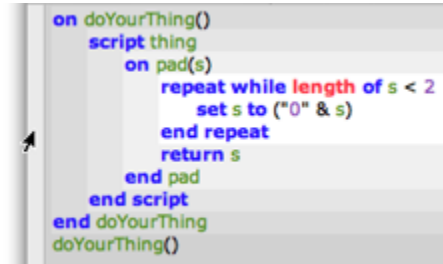
```
repeat
    -- your insertion point is at the start of this line
end
```

Now you're ready to type the content of the repeat block. This works also for lines beginning with `tell`, `on`, `script`, `if`, `try`, `using`, `considering`, and `ignoring`. Script Debugger's auto-closing feature is intelligent; if there is already a corresponding end line, Script Debugger won't create one.

To reverse your auto-closing preference setting momentarily, hold down the Control key as you type Return. Thus, for example, if auto-closing is turned *off* and you type **Control-Return** in a block opening line, Script Debugger behaves as if auto-closing were turned *on* and creates a corresponding end line. If auto-closing is turned *on* and you type **Control-Return** in a block opening line, Script Debugger enters a return character and no more.

Block Viewing and Selection

To clarify the block structure of your script visually, you can hover the mouse in the [gutter](#) (the area to the left of the script text). Script Debugger outlines the nested block structure, starting with the block to the right of the mouse, in successively darker shades of grey.



```
on doYourThing()
  script thing
    on pad(s)
      repeat while length of s < 2
        set s to ("0" & s)
      end repeat
      return s
    end pad
  end script
end doYourThing()
doYourThing()
```

This feature is turned on through an [Editor preference](#):

- **Highlight block when mouse hovers in gutter.** If checked, this feature is turned on.

When this feature is turned on, you can also **select** a block. To do so, click the mouse in the gutter when you see the nested block structure outlined. If you click several times in rapid succession, then each click selects the next block outwards. For example, in the above illustration, clicking once would select the `repeat` block; clicking twice would select the `on pad` block; clicking three times would select the `script` block; clicking four times would select the outermost `on` block.

Block Selection (Balance)

Another way to select a block is to use Script Debugger's Balance command. To do so, choose Edit > Balance.

Each time you choose Edit > Balance, Script Debugger selects the block enclosing the current selection. Thus, if you choose Edit > Balance repeatedly, you select the block enclosing the insertion point, then the block enclosing that block, then the block enclosing *that* block, and so on.

(The Balance command starts by selecting delimited code, if the initial selection is between [delimiters](#).)



Script Debugger provides features that help you enter, view, and select pairs of delimiters. Such delimiters are opening and closing double-quotes, parentheses, square brackets, and curly braces (and, much rarer, chevrons).

Delimiter Entry (Auto-Pairing)

Auto-pairing is a text entry feature where, when you type the first of a pair of delimiters, Script Debugger automatically enters the second of the pair for you.

Auto-pairing is turned on through an [Editor preference](#):

- **Auto-pair delimiters** ([{ " ' }]). If checked, auto-pairing is turned on.

To test auto-pairing, make a new script and type a quotation mark. A second quotation mark is created for you, and the insertion point is positioned between them, ready for you to type the contents of a literal string.

Auto-pairing is intelligent. When you have finished typing the contents of the string, so that your insertion point is positioned up against the closing quotation mark, like this...

```
set s to "this is a test"
```

...try typing a quotation mark. Script Debugger does *not* enter a new closing quotation mark; it understands that the closing quotation mark you are typing is the one that already exists, and it behaves as if you had just typed *that* quotation mark, moving the insertion point beyond it.

Another convenient feature of auto-pairing is its behavior when text is selected. Normally, if text is selected and you type a key, the character entered by that key *replaces* the selected text. But if auto-pairing is turned on, then if text is selected and you type an opening delimiter, the opening and closing delimiters *surround* the selected text. Thus, for example, if `howdy` is selected and you type a quotation mark, you'll get `"howdy"`.

To reverse your auto-pairing preference setting momentarily, hold down the Control key as you type an opening delimiter. Thus, for example, if auto-pairing is turned *off* and you type **Control-[**, Script Debugger behaves as if auto-pairing were turned *on*: if there is an insertion point, Script Debugger types `[]` with the insertion point between them, and if text is selected, Script Debugger surrounds it with `[]`. If auto-pairing is turned *on* and you type **Control-[**, Script Debugger enters `[` just as if it were any old character.

Delimiter Checking As You Type

With this feature (independent of auto-pairing), when you type a right (closing) delimiter, Script Debugger looks backwards to make sure that it matches a corresponding left (opening) delimiter. If so, Script Debugger highlights the earlier delimiter momentarily. If not, Script Debugger beeps.

This feature is turned on through an [Editor preference](#):

- **Auto-highlight opening ([{ when typing closing }]).** If checked, this feature is turned on. Use the slider to set how much delay there should be before the opening delimiter is highlighted, and the **Scroll if necessary** checkbox to set whether Script Debugger should momentarily scroll backwards if needed to reveal an opening delimiter.

Delimiter Selection (Balance)

To **select everything within a pair of delimiters**, click or select anywhere between the delimiters and choose Edit > Balance. An [Editor preference](#) determines whether the delimiters themselves will be included in the selection:

- **Balance includes enclosing ([{ }]) delimiters.** If checked, the Edit > Balance command selects everything including the surrounding delimiters; otherwise, it selects everything within the delimiters.

The Balance command also considers a comment to be delimited thing. Thus, if you choose Edit > Balance when the selection is within a single-line or multi-line comment, the entire comment will be selected.

This is actually the same feature used for [block](#) selection. If you choose Edit > Balance repeatedly, Script Debugger will select everything within delimiters, then everything within any delimiters surrounding those, and so on; then it selects the containing AppleScript code block, then the block containing that, and so on.

Text Completion

As you type the start of an AppleScript term, Script Debugger can fill in the rest of the term for you. This is essentially the Complete feature that appears in many other Cocoa applications, such as TextEdit. To **see a list of possible completions**:

- Type the beginning of an AppleScript term, then press Esc or choose Edit > Complete. A list of possible completions appears. (This list is drawn from scripting additions, AppleScript itself, and the [tell context](#), along with relevant identifiers in the script itself.) Or, if there is just one known completion, the completion is simply entered for you and selected.

For example:

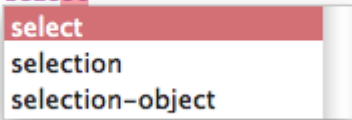
```
tell app "iTunes"
    tell bro|
end
```

At this point you press Esc and browser window is filled in for you, because it is the only known completion. Continuing:

```
tell app "iTunes"
    tell browser window 1
        get sele|
    end
end
```

At this point you press Esc and a list of possible completions appears:

```
tell app "iTunes"
    tell browser window 1
        get select
    end
end
```



When the list appears you can navigate it with arrow keys. The Return key (and many other possible keys) will accept a selected completion, or you can double-click an entry in the list. The Esc or Delete key closes the list without inserting a completion.



Text Substitution



You can define abbreviations with corresponding substitutions; when you type an abbreviation, Script Debugger replaces it with the substitution. This can be a convenient way to enter boilerplate text, commonly used control structures and commands, or characters that are difficult to type directly; you might also use it as a form of automatic spelling correction.

This feature is turned on through a [preference](#), and managed in the same preference pane:

- **To turn on text substitutions**, check **Automatic Substitution: Enabled** in the [Text Substitutions](#) preference pane.
- **To use a text substitution**, once this feature has been turned on, just type. For example, in a script, type `teh` and a space character; Script Debugger corrects `teh` to `the`. The rule is that when you type a sequence of characters, then when you type a following non-word character, such as a space, a tab, a Return, a comma, and so on, Script Debugger looks to see whether the preceding characters are an abbreviation; if they are, it automatically replaces them with the substitution.
- **To manage text substitutions**, edit the list in the [Text Substitutions](#) preference pane. Use the **+** button to create a new substitution; use the **-** button to delete a selected substitution. Double-click in the Replace or With column to edit text. Even if text substitutions as a whole are turned on, you can disable an individual substitution by unchecking it in the On column.

What you are managing here are text pairs. The Replace column is the text you will type (the abbreviation); the With column is the text that Script Debugger will replace it with (the substitution).

Script Debugger ships with some text substitutions included, and they illustrate various uses of this feature. For example:

- You can type `dd` (plus space) to get `display dialog "message"`, a convenient way to enter a commonly used command.
- You can type `!=` (plus space) to get `≠`; this corrects a common error (many languages use `!=` as the not-equal operator, but AppleScript does not) and also helps enter a difficult character (many users can't remember how to type `≠`).
- You can type `teh` (plus space) and get `the`; this corrects a common typo.

Observe that the substitution for `dd` includes the phrase `[select:message]`. This works just as for [clippings](#): it means that when this substitution is performed, the word `message` will appear and will be selected. This is very convenient because it means you can type rapidly and continuously. To enter `display dialog "howdy"` you just type `dd[space]howdy` and the right thing happens.

The substitution `idd` demonstrates that text substitutions can consist of multiple lines of code. (Try it in a script window!) If you find that the preference pane is not a convenient place to edit a multi-line text substitution, edit the substitution in a script window or a word processor and paste it into the With column.

Text substitutions are quite analogous to [clippings](#); indeed, they can be thought of as a convenient way to enter clippings. Text substitutions use the same [expansion tags](#) as clippings. Like clippings, text substitutions constitute boilerplate text that you're entering with a single command. With a clipping, that command is a menu choice, or its keyboard shortcut. With text substitution, the command is the abbreviation, which you type directly into your script. In many cases, an abbreviation is easier to remember or to type.



Miscellaneous Typing and Selection



This page collects some miscellaneous typing and selection features that you might otherwise be unaware of.

To **navigate with the keyboard**, Script Debugger supports a full repertoire of keyboard navigation shortcuts:

- Right Arrow and Left Arrow navigate by character.
- Option-Right Arrow and Option-Left Arrow navigate by word.
- Command-Right Arrow and Command-Left Arrow navigate to the two ends of the current line.
- Option-Up Arrow and Option-Down Arrow navigate a line at a time.
- Command-Up Arrow and Command-Down Arrow navigate to the start and end of the script.

To **select with the keyboard**:

- Add Shift to the above keyboard navigation shortcuts.

To **select a line**:

- Triple-click on the line. You can triple-click-drag to select a stretch of lines.
- Click to the left of the line. Just to the right of the [gutter](#) is a narrow area where this click works to select a line. You can click-drag to select a stretch of lines.
- Click or select anywhere within the line and then type Shift-Command-Right Arrow, Shift-Command-Left Arrow. You can then use Shift-Up Arrow or Shift-Down Arrow to select a stretch of lines.

To **start a new line**:

- Press Command-Return. No matter where the insertion point or selection is in the current line, it will be abandoned and a new line will be inserted below the current line, with the insertion point at its start, ready to type. To append an AppleScript line-continuation character to the current line at the same time, press Command-Option-Return. This feature may be combined with [auto-closing](#); for example, if you start in the middle of a `tell` line and press Command-Return, then if there is no corresponding end line, one will be created if auto-closing is turned on.



Script Debugger gives you many ways to **create a tell block** targeting a particular application. You can insert a tell block into an existing script, or you can create a new script and insert a tell block into it, in a single move. If text is already selected, the inserted tell block will enclose it (no text will be destroyed).

These are valuable shortcuts because entering an application's name manually at the start of a tell block is a tedious and error-prone operation. Also, most of the time, as you start to write a script you will already have in mind some application that you want to target, so creating a tell block is usually a better way to create a new script window than File > New Script!

- In a [script window](#), choose Edit > Paste Tell. (Alternatively, hold down the Control key and use the script window's contextual menu.) The hierarchical menu lists all running scriptable applications and all [previously encountered applications](#). Choose an application to insert a tell block targeting it.
- In the [dictionary window](#) for an application, choose Dictionary > Paste Tell (or click the Paste Tell button in the dictionary window's toolbar), to insert a tell block targeting that application.
- In the [Known Applications inspector](#), select an application and click the Paste Tell button.

In all of the above cases (except when using the contextual menu), if you hold down the Option key, a new window is created. Otherwise, you're inserting into the frontmost script window. If there is no open script window, a new window is created.


- Drag-and-drop an application from the Finder into your script window. A dialog appears asking what you want to do. One option is to paste a tell block targeting that application.

(Alternatively, drag-and-drop an application's name from the [Known Applications inspector](#) into your script window.)

- Type `ta` followed by a space. If [text substitutions](#) are turned on, a tell block targeting the Finder is created, and the word "Finder" is selected, ready for you to type the name of a different application.

Clippings

Script Debugger features **intelligent clippings** — bits of boilerplate text that you can insert into your code in a smart way. You can insert a clipping in any of the following ways:

- Choose it from the Clippings menu ().
- Choose Window > Inspectors > Clippings to show the Clippings inspector (if necessary), and then, in the Clippings inspector, double-click a clipping (or select it and click the Paste button).
- Control-click in a script window's text area and choose Paste Clipping from the contextual menu.

For example, a repeat block is a common control structure. Select some text in your script that you'd like to be inside a repeat block. Now choose a repeat block clipping, such as "repeat n times". A repeat block is inserted into your script window, wrapped around the text that you selected.

The tooltip for a Clippings menu item is the content of the clipping (hover the mouse over a menu item to see it).

If you're interested in creating your own clippings, [read on](#).

Further Details:

[How Clippings Work](#)



How Clippings Work



Script Debugger comes with [clippings](#) that correspond to all the commonly used AppleScript control structures. You can also add your own clippings, to implement any boilerplate that you frequently use. The clippings are text files in `~/Library/Application Support/Script Debugger 4.5/Clippings`, and you are free to add text files here.

Alternatively, you can keep clippings in the top-level `/Library/Application Support/Script Debugger 4.5/Clippings`. Yet a third possibility is to keep them in a folder called *Clippings* in the same folder as the Script Debugger application, but this option is mostly for backwards-compatibility and is not recommended.

A file will appear as a menu item. A folder will appear as a hierarchical menu, and the files inside it will be its menu items. The name of a file (or folder) is the name that will appear in the menu, except that certain names or part-names are hidden and used for determining the order of the menu, as follows:

- If a name starts with the prefix `##`, where `##` is a two-digit number (00-99), these digits are used to determine the position of this item in the menu and the prefix does not appear in the menu item's name.
- A name `##)–***` will appear as a menu separator, again with its order determined by the two-digit number `##`.

To **edit a clipping**:

- Choose it from the [Clippings menu](#) (or double-click it in [Clippings inspector](#)) while holding down the Option key.

To **reveal a clipping file** in the Finder:

- Choose it from the [Clippings menu](#) (or double-click it in [Clippings inspector](#)) while holding down the Shift key.

In the Clippings inspector you can also choose Reveal in Finder from the tool menu at the upper right.

A clipping's text is pasted literally into your script, except for the following expansion tags which are interpreted intelligently:

`[[selected-lines:default text]]`

This tag expands to the complete lines containing the script's current selection. If the current selection is just an insertion point, *default text* is used.

`[[selection:default text]]`

This tag expands to the script's current selection. If the current selection is just an insertion point, *default text* is used.

`[[select:text]]`

This tag expands to *text* and also selects it, ready for further typing that modifies the selection.

[[user]]

This tag expands to the user's full name, as shown in the Accounts preference pane.

[[account]]

This tag expands to the user's short name, as shown in the Accounts preference pane.

[[date:format]]

This tag expands to the current date and time, where *format* is an unquoted [strftime\(\)](#) format string. Alternatively, just say **[[date]]** and a standard format (mm/dd/yy hh:mm:ss) will be used.

[[VARNAME]]

This tag expands to the *VARNAME* environment variable. So, for example, **[[SHELL]]** would be expanded to something like `/bin/bash`. Anything in double brackets that doesn't match one of the preceding tag types is taken to be the name of an environment variable. If there is no matching environment variable name, the tag is left unexpanded. So, for example, **[[howdy]]** becomes `[[howdy]]`.

So, for instance, consider this clipping:

```
[[selected-lines:]]  
display dialog "[[select:howdy]]"
```

What does it do when pasted into your script? First, it skips past all lines containing the current selection, and inserts itself after the last of those lines, thus starting a completely new line. That new line says `display dialog "howdy"`, and the word `howdy` is selected so that you can now type a replacement string inside the quotation marks.



Inserting Content



Script Debugger has many convenient shortcuts for inserting content into your script from elsewhere. Here's a summary.

- You have many ways to [create a tell block](#) targeting a particular application.
- You can type an abbreviation corresponding to a [text substitution](#).
- You can type the start of an AppleScript term and let Script Debugger [complete it](#) for you.
- You can take advantage of [auto-closing](#) to enter end lines automatically, and [auto-pairing](#) to enter closing delimiters automatically.
- You can [paste a clipping](#) to insert boilerplate, such as an AppleScript control structure.

From the **Finder**, drag-and-drop a file or folder into your script. A dialog appears asking what you want to do.

- If what you dragged is an application, one option is to insert a [tell block](#) targeting that application.
- You can insert the name, alias, or pathname (POSIX or Macintosh-style) of the dropped items. If you dropped multiple items, these are placed in an AppleScript list.
- You can insert an object specifier (reference) suitable for use in a tell block targeting the Finder.
- If what you dragged is a text file, you can insert its contents.

From a [dictionary window](#), choose Dictionary > Paste Tell.

- If what's selected is a command, what's inserted is a template for issuing that command. The template is wrapped in a tell block if necessary.
- If what's selected is an event, what's inserted is a template for an event handler for that event.
- Otherwise, what's inserted is a tell block.

From an [explorer view](#), drag an entry into your script. (Drag from the first column.) Or, if this is a dictionary explorer, choose Dictionary > Paste Tell.

- What's inserted is a reference to the selected property or element (or element collection). The reference is wrapped in a tell block if necessary.

- Alternatively, if what you wanted to insert is the value of a property, control-click on the property and choose Copy Value (or hold Shift and choose Edit > Copy), and then paste into your script.



[Clippings](#)

[Shift](#)





You can remove or add a level of indentation to the selected lines. To **change the indentation level**:

- Select some text and choose Edit > Shift Left or Edit > Shift Right. (You can also add Shift Left and Shift Right buttons to the script window's [toolbar](#).)

The selection will be expanded to consist of complete lines, and a tab character will be removed from or added to the start of each selected line.

AppleScript will perform its own indentation when the script is compiled, which may alter the number of tab characters at the start of a line (except inside a multiline string literal).

See also on [Show Invisibles](#), [Show Spaces](#), and [Show Tab Stops](#).

Comment

Commenting and uncommenting code is a common need when developing AppleScript code, and Script Debugger provides an editing shortcut for doing this.

- To **comment out a stretch of code**, select the code and choose Edit > Comment. Script Debugger will extend the selection to consist of complete lines, and will then insert a single-line comment character at the start of each of those lines.
- To **turn comments into code**, select some code and choose Edit > Uncomment. Script Debugger will extend the selection to consist of complete lines, and will then remove a single-line comment character from the start of each of those lines, if there is one. (If there isn't one, that's fine. The line is left unaltered.)

(You can also add Comment and Uncomment buttons to the script window's [toolbar](#).)

Why does Script Debugger use single-line comments rather than surrounding the selected text with block comment delimiters, (`* like this *`)? One reason is that block comments are fragile. An unbalanced double-quote within block comment delimiters will keep your script from compiling. Single-line comments are simpler. In fact, with Script Debugger, multiple single-line comments are easier to deal with than block comments. To insert block comment delimiters, choose Clippings > Block Comment.

An [Editor preference](#) governs what is actually inserted at the start of each line when you choose Edit > Comment. This permits you to use, for example, either `--` (the traditional comment character) or `#` (the new comment character introduced in Mac OS X 10.5), and to set the number of spaces that should follow the comment character.



You can **convert tabs to spaces and vice versa**. To do so:

- Select some text and choose Edit > Entab or Edit > Detab. (You can also add Entab and Detab buttons to the script window's [toolbar](#).)

The selection will be expanded to consist of complete lines, and the indentation whitespace at the start of those lines will be converted to tabs or spaces respectively.

This is particularly useful when a script must be pasted into some other environment, where spaces are better interpreted than tab characters. For example, before copying and pasting a script into an email message or a web page, you might Select All and then Detab.

See also on [Show Invisibles](#), [Show Spaces](#), and [Show Tab Stops](#).

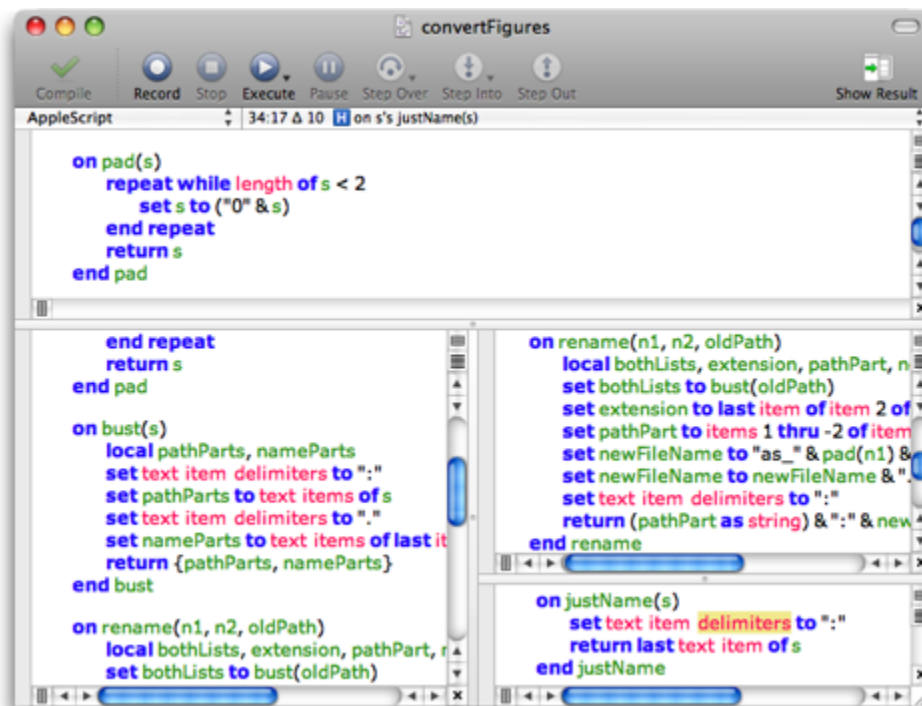





Splitting the Editor




In a long script, it can be useful to view and edit more than one area of your script simultaneously. For this reason, Script Debugger lets you split the [script window text area](#) into multiple panes. You can split the text area vertically or horizontally — and you can split each resulting pane vertically or horizontally. Each pane can be scrolled to display a different region of the script.

(But you'd probably never split a window into panes quite as insanely as in this picture!)



- **To split a script window or pane vertically**, click the vertical split icon () in the vertical scrollbar, or choose Edit > Split Editor Vertically.
- **To split a script window or pane horizontally**, click the horizontal split icon () in the horizontal scrollbar, or choose Edit > Split Editor Horizontally.
- **To resize a pane**, drag the divider line between panes.
- **To close a pane**, click its close icon () at the lower right corner of the pane, or choose Edit > Close Split View.

- **To close all panes**, Option-click any close icon () , or choose Edit > Close All Split Views.

What should happen when you change a [view](#) setting (such as [line wrapping](#) or visibility of [tab stops](#)) in a split pane? Should other split panes change to match, or should their view settings remain independent? Script Debugger lets you decide. An [Editor preference](#) lets you set the base behavior:

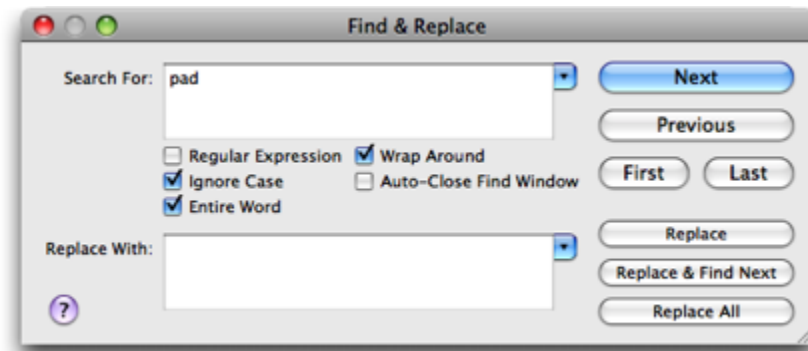
- **Synchronize split-view appearance.** If checked, then when you change a view setting in one split pane, other split panes of the same window will change the same setting to match.

Whichever way you choose, the Option key is a temporary toggle. Hold down the Option key as you change a view setting to make Script Debugger behave the opposite way from the base behavior preference. For example, if **Synchronize split-view appearance** is *not* checked, then if you hold down the Option key and change the [line wrapping](#) of a split pane (by clicking its wrap icon, or by choosing View > Wrap Lines), all split panes in this window will adopt the new line wrap setting.



Script Debugger provides excellent find (and replace) facilities.

You can operate from within the **Find dialog** (which appears when you choose Search > Find), or you can use the menu items in the [Search menu](#) (or their keyboard shortcuts). Most likely you'll settle on a combination of the Find dialog and a few commonly used keyboard shortcuts.



Checkbox options in the Find dialog affect subsequent searching even if the Find dialog is no longer showing. Thus, a common way to find is to choose Search > Find to bring up the Find dialog, enter the text to look for, set options, and press Return to click the Find button, and then use Search > Find Again to look for subsequent instances.

If you can see text, you can search for that text without bringing up the Find dialog at all. Using the Search menu items, you can enter selected text into the Search For field or the Replace With field, or you can use Search > Find Selection to enter selected text to the Search For field and find it, in a single move.

Checkbox options, as well as buttons, in the Find dialog are generally self-explanatory. Here are comments on less obvious features:

- To **search backwards** as you press one of the three Replace buttons, hold down the Shift key.
- The **Wrap Around** setting applies even to Replace All. If **Wrap Around** is unchecked and you do a Replace All, only instances of the Search For text *after* the insertion point will be affected.
- If **Auto-Close Find Window** is checked, then if you click the Next, Previous, First, or Last button and the search succeeds, the Find window will close.
- The little pop-down arrows to the right of the text fields summon **history lists** of Search For and Replace With terms that you've used earlier. Click a term to enter it conveniently into the text field.

- Script Debugger supports optional use of **regular expressions** in the text fields of the Find dialog. If you don't know about regular expressions, the best teacher is still [Jeffrey Friedl's book](#). Script Debugger uses the [ICU flavor](#) of regular expression syntax.

Experts: Observe that by default `.` matches any character including the return character at the end of a line; you can turn this off with `(?-s)`.

An [Editor preference](#) lets you set whether the Search For field contents are shared with other applications.



[Splitting the Editor](#)

[Go To](#)





Script Debugger has a “go to” feature that lets you **jump to a line by number**. To do so:

- Choose Search > Go To Line and enter a line number in the dialog. (The number that appears when you summon this dialog is the number of the line where you currently are.)

To work more easily with line numbers, you can elect to [show line numbers](#) in your script. Also, the [navigation bar](#) (at the top right, above the text area in the script window) always shows the current line number. And there’s an [Editor preference](#) that causes line numbers to be shown in a tooltip while you scroll.



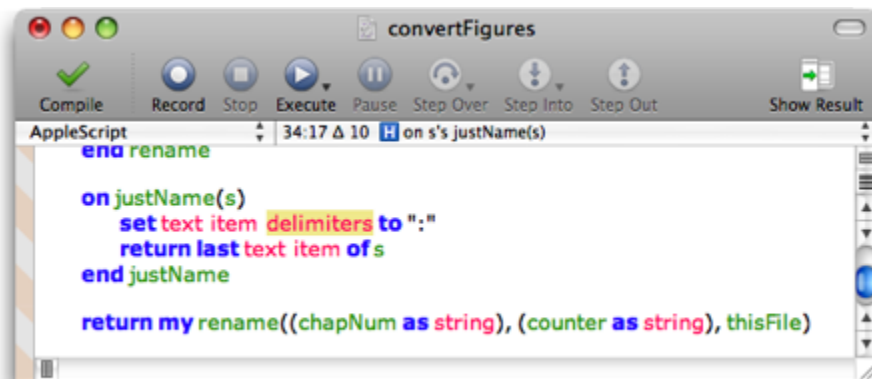
Script Debugger provides convenient ways to navigate through the structure of your AppleScript code.

As AppleScript programs become larger than a few lines, it's common practice to divide them into smaller blocks — handlers and script objects. You should adopt this practice, because it allows Script Debugger to help you navigate your code.

One possibility is to navigate by **jumping from handler to handler**, successively. To do so:

- Choose Search > Go to Next Handler and Search > Go to Previous Handler. These jump to the first line of successive handler definitions.

For more power, use the **navigation bar**. This is the rectangle at the top of the [script window](#), to the right, above the text area. The navigation bar shows you where you are, and it also contains the **table of contents** popup menu, which lets you jump easily to any part of your script.



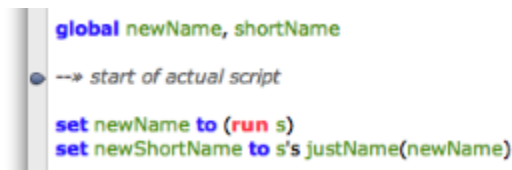
The above illustration shows a typical navigation bar display. 34:17 means the selection starts at line 34 of the script, and character 17 of that line. (It may look like character 15 to you, but there are two [tab characters](#) creating the indentation.) Δ10 means the selection is 10 characters long. The **H** symbol means you're in a handler (other possibilities are **S** for script object, **P** for script property declaration, and **G** for global declaration). Finally, the phrase at the end sums up your position in structural terms. Here, on s's justName(s) means we're in a script object s that contains a handler justName that takes one parameter (also called s), and we're inside that handler.

Hold down the mouse on the navigation bar to bring up the **table of contents menu**.

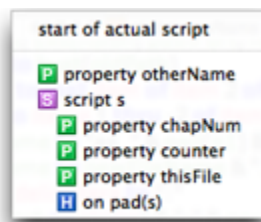


The menu shows all script object and handler definition blocks, along with all script property declarations and top-level global declarations, in an intuitive hierarchical layout. (Hold down the Option key while choosing the menu to show it without the properties and globals.) The check mark shows where the selection is now. Choose an item in the menu to jump to it.

You can also insert **markers** into your script. A marker's significance is that it appears in the table of contents menu, which means you can jump to it. A marker is defined as an AppleScript comment that starts with `-->>`. (If [text substitution](#) is turned on, this may be converted to `--»`, but that's okay; it's still a marker.) AppleScript ignores a marker (because it's a comment). But Script Debugger sees it, and lets you know this with a "droplet" symbol in the [gutter](#):



The text of the comment appears in the table of contents menu:



The order of items in the table of contents menu is determined by an [Editor preference](#) — either items are sorted alphabetically (**Sort menu** is checked) or they appear in the same order as they appear in the script (**Sort menu** is unchecked). Whichever order you prefer, hold down the Shift key while choosing the menu to show it sorted the other way.

An [Editor preference](#) also causes a tooltip to appear whenever you scroll your script. This tooltip shows the line number and navigation bar entry corresponding to the first line currently visible in the script window.



External Editor



Script Debugger lets you use an alternative text-editing application to edit a script — [BBEdit](#), [TextWrangler](#), or [TextMate](#). Script Debugger's implementation of this feature is seamless. With a [script window](#) open in Script Debugger, you open the same script in an external editor and edit it there. When you save and close the document in the external editor, you are back in the Script Debugger document, which has taken on the changes you made in the external editor.

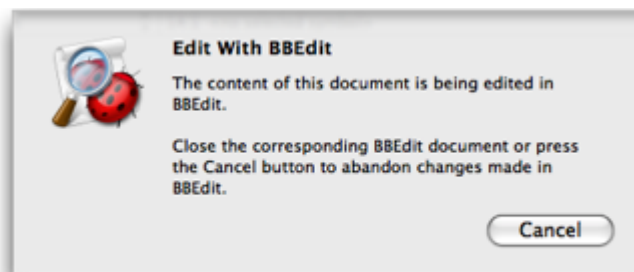
To **start an external editing session** on the frontmost script window:

- Choose File > Edit With BBEdit. Actually, the name of the application shown in this menu item will depend on which of the three applications you have and which ones are running. Script Debugger will prefer a running application to one that is not running, and among running applications or non-running applications it will prefer the order BBEdit, TextWrangler, TextMate.

Thus, for example, if you have both BBEdit and TextMate, this menu item will be called Edit With BBEdit if both BBEdit and TextMate are running or if neither is running, but it will be called Edit With TextMate if TextMate is running and BBEdit isn't.

When you start an external editing session, two things happen:

1. In the frontmost Script Debugger script window, a dialog appears warning that the script is being edited externally.



Additionally, the script window is watermarked in its lower right corner with the word "Locked".



2. A new window opens in the external editing application, containing the text of the Script Debugger script window (and the external editing application comes to the front).

At this point, the normal chain of events is that you would edit the document in the external editing application:

- Whenever you **save** the document in the external editing application, the Script Debugger copy is updated to match.
- When you **close** the document in the external editing application, the warning dialog is removed from the Script Debugger script window (and Script Debugger comes to the front). This is the normal way in which an external editing session ends in good order.

Alternatively, you might change your mind and decide to **break off the external editing session** prematurely without reflecting the changes from the external editing application back into the Script Debugger document. To do so:

- Switch back to Script Debugger and click Cancel in the warning dialog.



When you [compile](#) and [run](#) (or “execute”) your script, Script Debugger gives you tools for understanding what happened.

- You can view the [result](#) in powerful ways, find out [how long](#) your script took to run, and see the value of [persistent variables](#) afterwards.
- If there are [errors](#) during compilation or runtime, they are clearly displayed, with full supplementary information. You can also use the [Apple Event Log](#) to find out exactly what interapplication communications took place.
- If a scriptable application is also recordable, you can [record](#) it with Script Debugger.
- You can set a script’s [target](#) or [parent script](#).

Further Details:

[Compile](#)
[Execute](#)
[Result](#)
[Times](#)
[Variables](#)
[Errors](#)
[Apple Event Log](#)
[Record](#)
[Default Target](#)
[Parent Script](#)

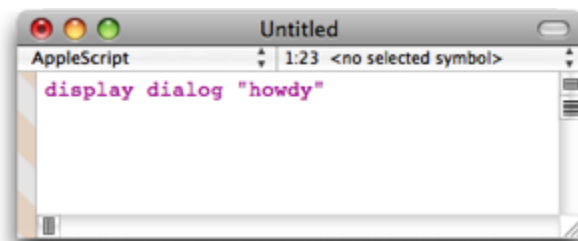


Compile



Before AppleScript will [execute](#) your code, the code must be compiled. You can compile your code separately, or else just try to run it (in which case, if it needs compiling, Script Debugger will automatically ask AppleScript to compile it before running it).

Any time your script is modified, it needs compiling. Script Debugger provides a clear indication of **whether your script needs compiling**. If you have checked the [Editor preference](#) “Show compiled state in gutter”, then a script that needs compiling has a striped “barberpole” pattern in the [gutter](#). The illustration below shows a script that needs compiling.



To **compile your code** without running it:

- Choose Script > Compile. Alternatively, you can click the Compile button in the [script window](#) toolbar, or press the Enter key (different from the Return key).

If your script can't be compiled because it isn't valid AppleScript code, you'll get an [error](#) message.

To **force** your script to recompile even if it hasn't been altered, hold down the Option key and choose Script > Recompile (or click the Recompile button in the toolbar). This will also [re-initialize persistent top-level entities](#) to their base values.





Script Debugger can run a script. In fact, it can run (and [debug](#)) multiple scripts simultaneously. To **execute (run) a script**:

- Choose Script > Execute (or click the Execute button in the toolbar).

If the script needs [compiling](#), Script Debugger attempts compilation first. If there is a compilation [error](#), the script won't start to run.

Otherwise, the script runs. While a script is running:

- A circular progress indicator spins at the right end of the title bar.
- Whenever an Apple event has been sent to an application but the application has not yet replied, the application's icon appears at the right end of the title bar.



- In the [toolbar](#), all buttons go dim except for the Stop button. You can click this, or choose Script > Stop, to **interrupt execution**.
- In the [Script menu](#), the Execute menu item changes to Running and has a check mark.
- In the [Window menu](#) (and the [Windows Inspector](#)), the listing for this script window is badged with an icon indicating that it is running.

Additionally, if the script is in [debug mode](#), then when the script is [executing](#), the [Pause button](#) is enabled (along with Script > Pause).

When execution ends, the circular progress indicator goes away, and the Stop button is disabled. If no uncaught runtime [error](#) was encountered — that is, if the script ran all the way to a natural conclusion — there is usually a [result](#).

When you start to execute a script, Script Debugger automatically saves a copy of the script in a private location. If execution of the script happens to crash Script Debugger (which can occur if a target application is particularly badly behaved), just start up Script Debugger again. Your script will be opened automatically, magically restored, just as it was when you started to execute it.

Note that you can control the execution of the frontmost script even when you're in a different application! Use Script Debugger's Dock menu. It contains Execute and Stop menu items (and others).

Script Debugger helps you run [individual handlers](#) in a script.

Further Details:

[Testing Handlers](#)



 [Compile](#)

[Result](#) 



Testing Handlers

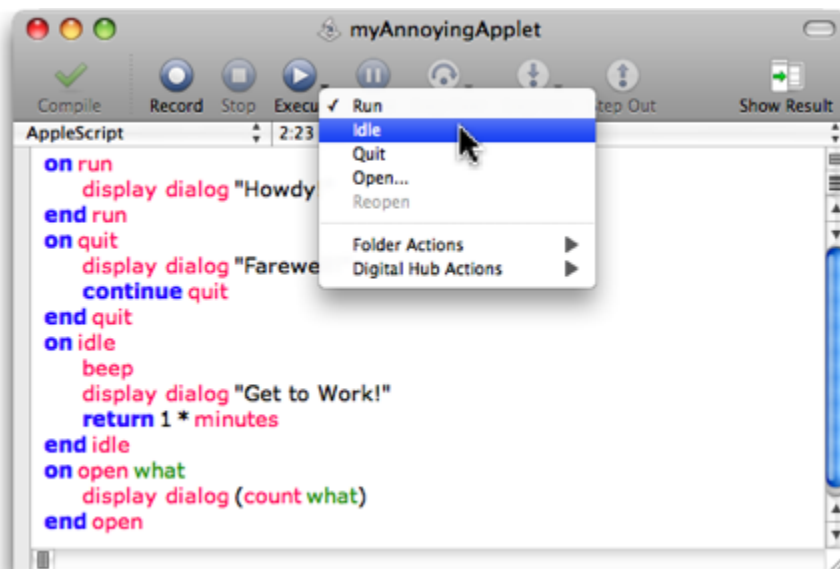


Script Debugger lets you run individual handlers in a script. This can be a valuable testing and debugging technique.

Standard Event Handlers

Consider the [applet](#) script in the illustration below. If you simply run this script, you'll see the "Howdy!" dialog. That's because what you've just run is the run handler. But what if you want to test the `idle` handler? Script Debugger lets you do this. To do so:

- Choose Script > Execute > Idle. Alternatively, hold down the mouse down on the Execute button in the toolbar. A popup menu appears, and you can choose Idle in this menu.



This does two things:

1. It actually runs the `idle` handler. (To prevent this, hold down the Shift key as you choose Idle.)
2. It sets the "default" handler of this script (also called the script's **current event**) to be the `idle` handler.

The menu you are using here — the menu that is attached hierarchically to Script > Execute, and that appears when you hold down the mouse down on the Execute button — is called the **Event Handler menu**. It is attached to Script > Execute, Script > Trace, Script > Step Over, and Script > Step Into, as well as to the Execute, Step Over, and Step Into toolbar buttons.

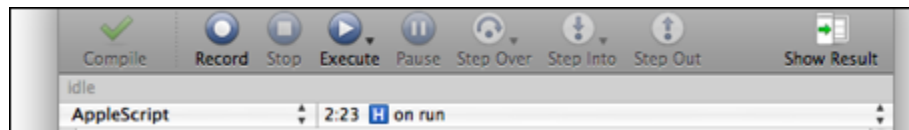
The standard event handlers listed in these menus are the five standard applet events, the Folder Actions suite events (defined in *StandardAdditions.osax*), and the Digital Hub Actions suite events (defined in *Digital Hub Scripting.osax*).

Note: If your script has one of these event handlers, but the corresponding menu item is not enabled, [compile](#) the script. That should fix it.

The Current Event

What does it mean to say that the script's current event is the `idle` handler? It means that the *next* time you execute the script, you'll execute the `idle` handler *automatically*. This behavior is there to make it easy for you to test an event handler, make a change in the script, and then test the event handler again. But it involves a change in the basic behavior of your script, since normally when you run a script, it's the run handler that runs.

To alert you to the fact that there is a current event, the current event is checked in the menu, and a **current event indicator** appears below the toolbar of your script window:



To revert the current event to being the run handler once again, choose Script > Execute > Run (again, hold down the Shift key to prevent the run handler from running when you do this). The current event indicator goes away.

Parameters and the Event History

The open handler and the event handlers defined in the Folder Actions and Digital Hub Actions suites are a bit different, because they expect parameters, which are aliases or files. For example, an open handler expects a list of aliases to the files and folders. And an `adding folder items to` handler expects two parameters, an alias to the watched folder and a list of aliases to the added files. Script Debugger helps you even further with this sort of handler.

Such a handler is listed in the Event Handler menu with an ellipsis (...) after its name. When you choose such a handler from the menu, Script Debugger puts up a dialog where you can select files and folders. An appropriate parameter or parameters will then be passed to the specified handler.

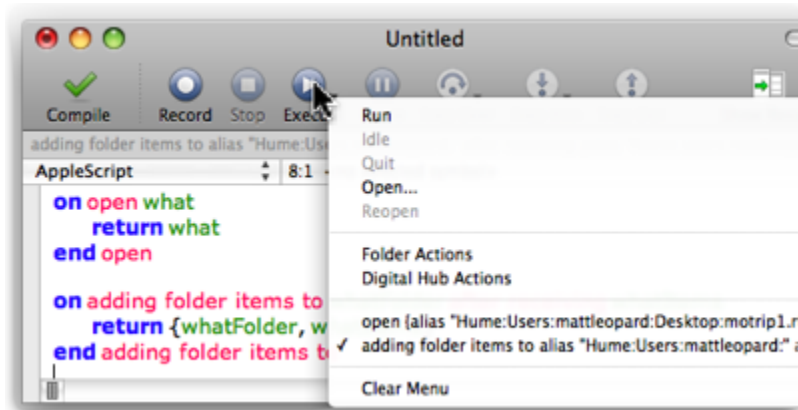
In the case of `adding folder items to` and `removing folder items from`, what you're supplying in this dialog is the *second* parameter, the item(s) that are allegedly being added or removed. Script Debugger will then use the containing folder of the item(s) as the first parameter, the watched folder.

In the case of just the open handler, there is another alternative. Drag-and-drop files and folders directly from the Finder into your script. If your script has an open handler, one of the options in the resulting dialog is to invoke the open handler with these Finder items as parameter.

Script Debugger remembers each alias or list of aliases produced in this way, along with the event handler it is to be passed to. Script Debugger adds this information to the bottom of

the Event Handler menu, and makes it the current event. This list of remembered event handlers and parameters is called the **event history**.

Thus, the next time you want to test *this handler with these same parameters*, you just choose it from the event history (or, if it is the current event, click the Execute button). And of course you can switch from testing one handler/parameter set to testing another, by choosing that menu item from the event history.



The event history is remembered until you close the script window; or you can deliberately remove it by choosing Clear Menu (which is always the last item in the Event Handler menu if there is an event history).

A `continue` statement in an event handler called in this way will generate an error in Script Debugger. This is deliberate and may be safely ignored. The technical reason is that if, for example, we permitted the `continue quit` statement in your quit handler to execute, Script Debugger itself would quit!

Other Handlers

What if the handler you want to test is *not* one of the standard event handlers already listed in the Event Handler menu? No problem. You can still call just the specific handler, and it is added to the event history so that you can easily call it again with the same parameters. To do so, you take advantage of Script Debugger's [scriptability](#), which allows one script to call a handler in another script.

Actually, you can do this for standard event handlers as well. For example, in the case of the open handler discussed above, you could open a new script window in front of the applet script (this ordering is to make sure that the applet script is document 2) and, in the new window, run this script:

```
tell application "Finder" to get disk 1 as alias
tell document 2
    open result
end tell
```

This calls the open handler in the applet script and adds it, with this parameter, to the applet script's event history.

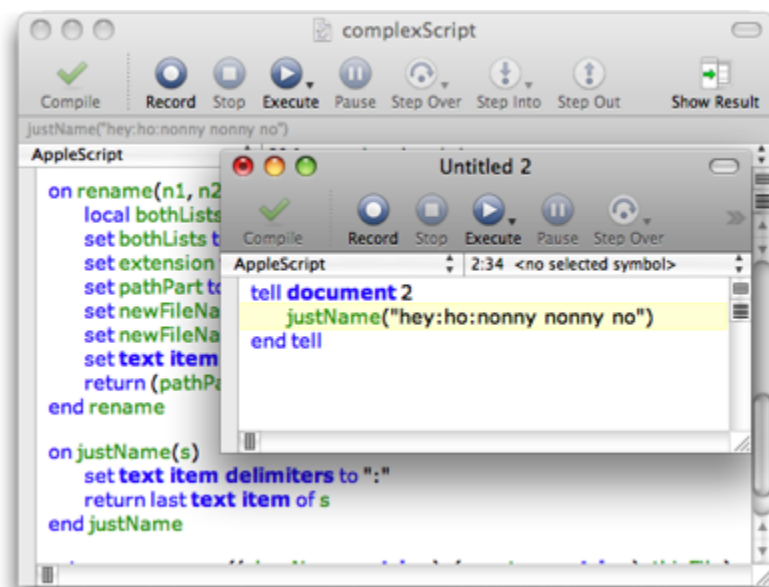
However, you are more likely to use this technique to test a top-level user handler. For example, suppose you have a complex script containing this top-level handler:

```
on justName(s)
    set text item delimiters to ":"
    return last text item of s
end justName
```

You can check whether this handler is behaving correctly, without the inconvenience and overhead of running the rest of the complex script. Make a new empty script window in front of the complex script, and enter and run this script:

```
tell document 2
    justName("hey:ho:nonny nonny no")
end tell
```

This calls `justName` in the complex script, and adds the call with this parameter to the event history. Now you can repeatedly test this handler and develop it without altering the rest of the complex script.



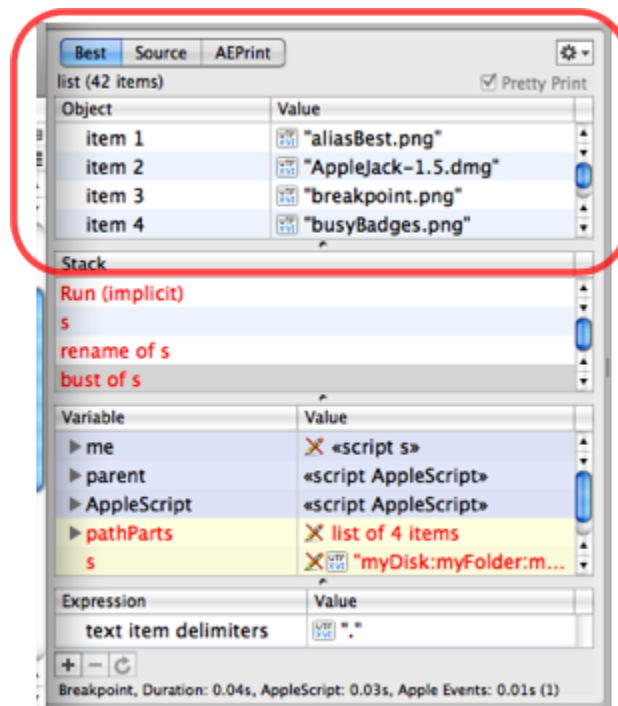
Similarly, during [external debugging](#), an entry-point handler called by the external application is added, with its parameters, to the target script's event history. (This happens only if the script is in debug mode, though, since otherwise this would not be [external debugging](#).)



Result



Most scripts, when [run](#), will implicitly or explicitly generate a result. This result is entered into the script window's **result drawer**, in the top pane (the **result pane**) of the drawer.



A [Debugger preference](#) lets you set what happens when a script completes running ("Show result when scripts pause or end"):

- **No.** If the drawer is not open, it does not open automatically. To summon the drawer manually if it isn't open, choose Script > Show Result (or click the Show Result button in the toolbar).
- **Show Result Drawer.** The drawer opens, if necessary, revealing the result. This is the factory default setting, and will probably best suit most people's way of working. Seeing the result drawer has certain advantages. Not only do you see the result, but you also view [persistent variables](#) and [execution times](#).
- **Show Result Viewer window.** The result is displayed in a separate [viewer window](#) — in fact, it is the very same viewer window you can summon manually by choosing Script > Show Result in Viewer.

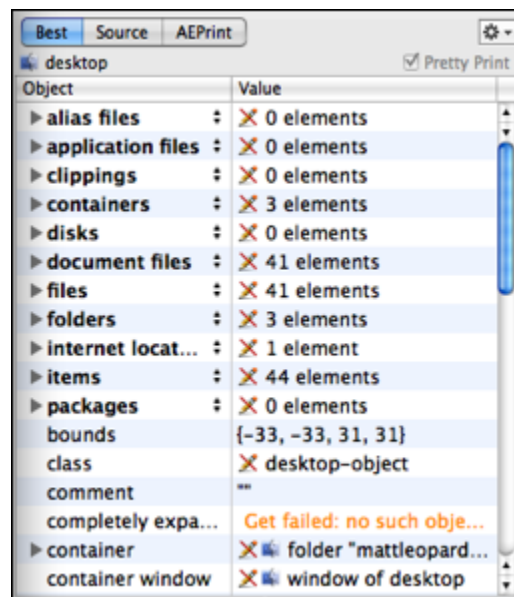
An [Editor preference](#) lets you set whether the open or closed state of the result drawer should be remembered when a script is saved and restored when it is next opened.

The open or closed state of the result drawer is one of the window features that you can configure by setting the [default state](#) for all new script windows.

The result pane is a [viewer](#), with many capabilities and [view options](#). In particular, when the result is an object reference, the result pane in [Best view](#) is an [explorer view](#). Script Debugger probes this object reference to obtain its elements and properties and their values *when the result is generated*. This feature is very informative and can reduce your script development time. So, for example, consider this script:

```
tell application "Finder"
    get desktop
end tell
```

The result is the reference desktop of application "Finder", but in [Best view](#) the result pane shows you more than this — it shows you *all about* the desktop at this moment:



Object	Value
▶ alias files	✗ 0 elements
▶ application files	✗ 0 elements
▶ clippings	✗ 0 elements
▶ containers	✗ 3 elements
▶ disks	✗ 0 elements
▶ document files	✗ 41 elements
▶ files	✗ 41 elements
▶ folders	✗ 3 elements
▶ internet locat...	✗ 1 element
▶ items	✗ 44 elements
▶ packages	✗ 0 elements
bounds	{-33, -33, 31, 31}
class	✗ desktop-object
comment	""
completely expa...	Get failed: no such obje...
▶ container	✗ folder "mattleopard...
container window	✗ window of desktop

Moreover, since this is an explorer, you can [do all the things](#) in the result pane that you can do in any explorer. Not only can you separate off the result pane *itself* as an individual viewer window (by choosing Script > Show Result in Viewer); you can separate off any entry *within* the result as an individual viewer window. You can drill down the hierarchy within the explorer, you can transfer references and values from the explorer to your script, you can ask for a dictionary definition, and you can even alter values in real time, thus affecting the running target application.

Further Details:

[Viewer](#)



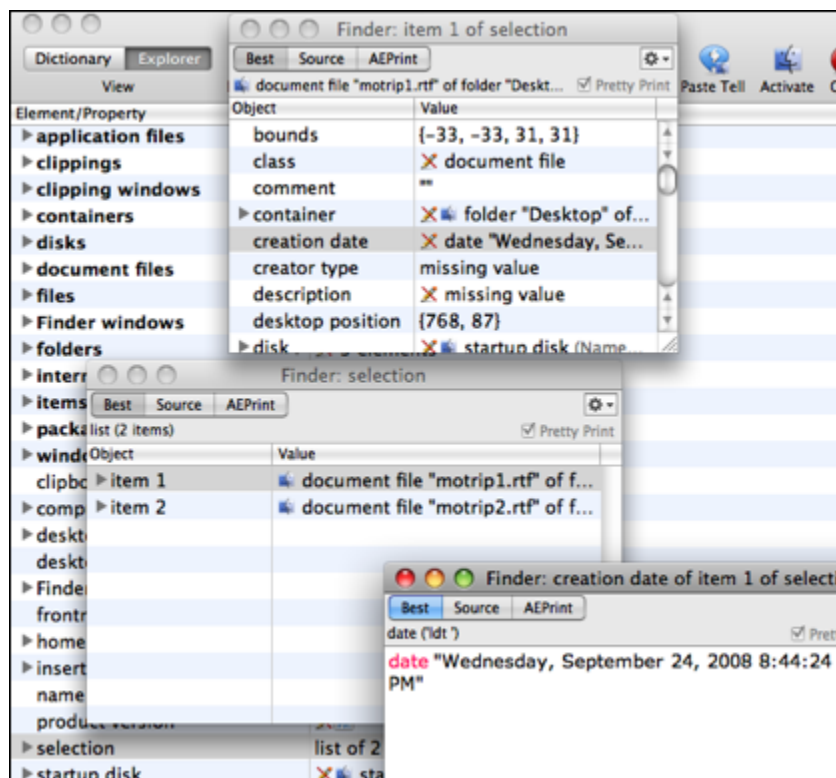
Viewer



A **viewer** is Script Debugger's powerful, flexible way of displaying an AppleScript value. Viewers appear, or can be generated, in several parts of the Script Debugger interface. The [result](#) of running a script initially appears as a viewer pane in the script's drawer, aspects of an [error](#) from running a script may appear in a viewer pane, and double-clicking a line of an [explorer view](#) opens a separate viewer window.

A viewer pane, such as the [result pane](#), can itself be reopened as an individual viewer window. This can be a very convenient thing to do, especially when the result pane contains a lot of information. To separate off a result pane off as an individual window, choose Script > Show Result in Viewer. (Alternatively, use the "tools" popup menu at the upper right of the result pane. Choose the Show Result in Viewer item.)

A separate viewer window, or a [cascade](#) of separate viewer windows (as shown in the illustration below, which shows three viewer windows in front of an explorer view), can be a way to focus more easily on the information that interests you.



If there is a change in the data from which a viewer window was generated, the viewer window changes. For example, you can display a script result as a separate window and leave that window open. Every time you run your script, this separate result viewer window will change to show the new result. Similarly, if you open a separate viewer window from an

explorer and then reload the data in the explorer, the viewer window will change (if necessary) to reflect the changed data.

A viewer window may also close spontaneously if the value being viewed ceases to exist. For instance, a viewer window that is viewing a variable from the [variables pane](#) will close when the variable goes out of scope. A viewer window that is viewing an element of a class will close if that class is refreshed and the element no longer exists.

Viewer Options

Viewer windows and viewer panes have many options and capabilities.

You have a choice of three views — [Best](#), [Source](#), and [AEPrint](#). You can toggle [pretty-printing](#) on and off.

There are also various other options for how material is displayed in each view. These are essentially the same as certain [display options](#) for [script windows](#). Where appropriate, you can toggle [wrapping](#), and you can show [tab stops](#), [invisible characters](#), and [spaces](#).

To **set defaults** for the **size, view, and display options** of viewer windows:

- Summon a viewer window, set its size and display as desired, and choose Window > Set Default Viewer Size & State.

There are two menus within the viewer: the **“tool” popup menu** at the upper right of the viewer (it has an icon like a gear), and the **contextual menu** (control-click to summon it). They bring together the above options, along with some additional capabilities. You can look up a class or native datatype in the dictionary. If what’s being viewed is an alias or some other reference to a file on disk, you can reveal or open the corresponding item in the Finder.

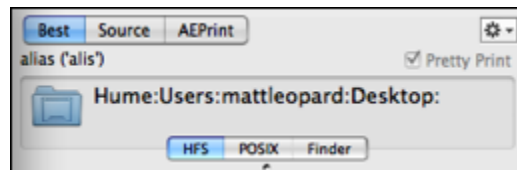
Further Details:

[Best](#)
[Source](#)
[AEPrint](#)
[Pretty-Print](#)



Best view in a [viewer](#) is significant in the following cases:

- If the value being viewed is an **object reference**, the actual object's elements and properties are shown, in real time, as an [explorer view](#).
- If the value is a **collection** (a list or a record), the items of the collection are displayed in an explorer.
- If the value is a valid **alias or a file object reference**, it is shown as a pathname that can be displayed in different styles (HFS or colon-delimited, POSIX or slash-delimited, and as the Finder would refer to it), as shown in the illustration below.



- If the value is **image data**, it is shown as an image.

Otherwise, you won't see much difference from [source view](#).



Source view in a [viewer](#) presents the value as the source language (usually AppleScript) would present it. This is particularly telling in cases where it can be contrasted with [Best view](#). For example:

- In Best view, an object reference is an [explorer view](#). In Source view, an object reference is the reference.
- In Best view, a list is shown as an explorer. In Source view, a list is delimited by curly braces.
- In Best view, a string is shown as the text of the string. In Source view, a string is delimited by quotation marks.



AEPrint



AEPrint view in a [viewer](#) shows the value as it would be communicated through an Apple event. For example, here's the Finder's desktop in AEPrint view:

```
'obj' '{
  'form': 'prop',
  'want': 'prop',
  'seld': 'desk',
  'from': 'psn '($000000000000C0001$)
}
```

In the [Apple Event Log](#) window, extra 'ascr' events are also generated corresponding to the points at which the tell target changes.

```
'ascr'\ 'tell' {
  '----': 'psn '($000000000000C0001$)
}
'core'\ 'getd' {
  '----': 'obj' '{
    'form': 'indx',
    'want': 'cwin',
    'seld': 1,
    'from': 'null'()
  },
  &'csig': 65536
}
'ascr'\ 'tend' { }
```

If you're not someone who knows or cares about AEPrint format and raw Apple events, then don't worry about AEPrint view.



Pretty-Print



Pretty-printing in a [viewer](#) arranges text in lines, with indentations, to make the value easier to read. Pretty-printing is useful in [Source](#) view and [AEPrint](#) view. To give a simple example, pretty-printing makes the difference between this (no pretty-printing):

```
{1, 2, 3}
```

and this (pretty-printing):

```
{  
    1,  
    2,  
    3  
}
```

To **control pretty-printing**:

- Choose View > Pretty Print. If the menu item is checked, pretty-printing is turned on. Alternatively, use the Pretty Print checkbox within the viewer. (And the [Apple Event Log](#) window has a Pretty Print button in the toolbar.)



[AEPrint](#)

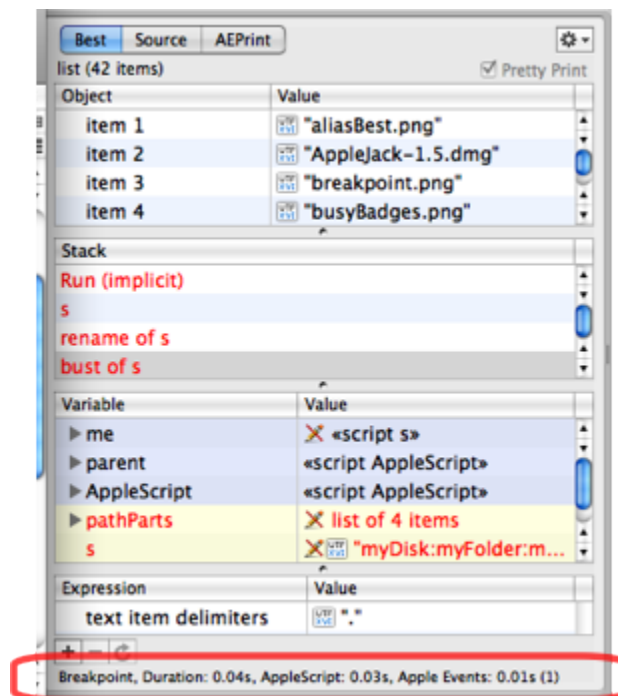


Times



Script Debugger automatically **times your script** as it runs. The timings accumulate as the script runs, and the total cumulative results can be viewed afterwards. This feature is especially useful during development when you're trying to make your script as fast and efficient as possible.

To **learn how long your script took to run**, you don't need to do anything special. Just [run](#) the script. The [result](#) drawer displays the timings, at the bottom of the drawer:



A typical timing might be something like this:

Duration: 1.12s, AppleScript: 0.03s, AppleEvents: 1.09s (92)

The times displayed consist of the total duration, followed by a breakdown into time spent within the script itself and time spent sending Apple events (and waiting for the replies). The two latter times will sum to the former. The number in parentheses is the number of Apple events sent.

You can also get a handle on what Apple events are being sent by using the [Apple Event Log](#).

◀ [Result](#)

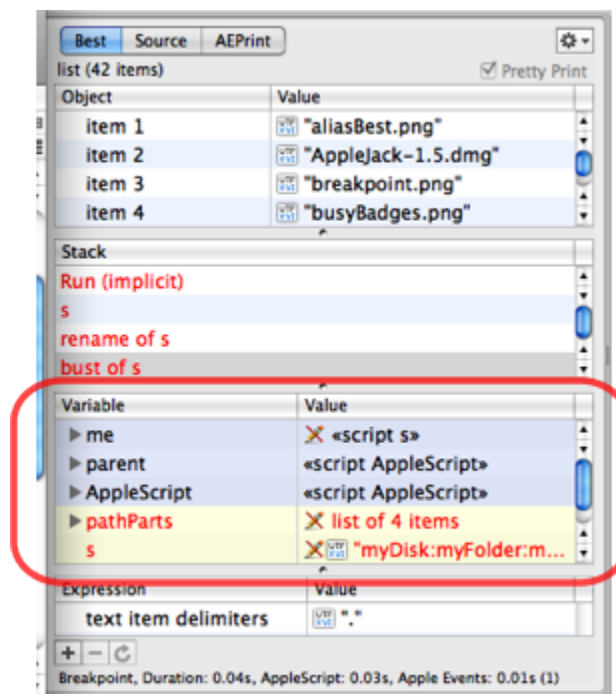
[Variables](#) ▶



Variables



Script Debugger displays the names and values of persistent variables (top-level entities — script objects, script properties, and globals) contained in your script. They are shown in the [result](#) drawer, in the third pane, whose columns are headed “Variable” and “Value”. This is the **variables pane**.



For example, running this script:

```
property x : missing value
set x to "Hello there!"
```

results in the variables pane looking like this:

Variable	Value
parent	«script AppleScript»
AppleScript	«script AppleScript»
x	"Hello there!"

The notion “contained in” is made more complicated by AppleScript’s script object inheritance mechanism. By default, your script’s parent is the AppleScript scripting component, so your script effectively “contains” the top-level entities of this virtual script object — and Script Debugger displays it. In the above illustration it displays

it twice, once by virtue of being the script's parent, and again by virtue of being the named virtual script object `AppleScript`, which is always globally accessible.

The variables pane is an [explorer view](#), with [all that this entails](#). For example, you can double-click a line of the variables pane to see that value displayed in its own [viewer window](#). You can also edit (change) the value of persistent top-level entities shown in the variables pane (for example, in the above illustration, the property `x` is editable); you might use this feature to experiment with your script's behavior under different initial values of your persistent entities. (Script Debugger [preserves](#) the value of persistent top-level entities when a script is saved and re-opened.)

The variables pane has a more [extended role](#) during [debugging](#). It then displays not only persistent top-level entities, and not only after the script has run — it displays *all* variables, while the script is still in the process of running.



AppleScript generates two kinds of error — compile errors and runtime errors — and Script Debugger provides full information about where the error occurred and what went wrong.

Script Debugger presents error information in a dialog. When you dismiss the dialog, you'll find that your script has been clearly marked with the **location of the error**, in three ways:

- A little "stop sign" icon (❌) for a compile error, or "red arrow" icon (➡) for a runtime error, appears in the [gutter](#) next to the problematic line.
- The line itself is highlighted, and the troublesome words are selected.
- A little red mark appears in the scroll bar.

To **show the error message again** after dismissing it:

- Choose Script > Show Last Error, or click on the "stop sign" icon (❌) or "red arrow" icon (➡) in the gutter. Alternatively, hover the mouse over the stop sign or red arrow, to see the text of the error message in a tooltip. (You can also add a Show Last Error button to the script window's [toolbar](#).)

To **scroll to where the problem is**:

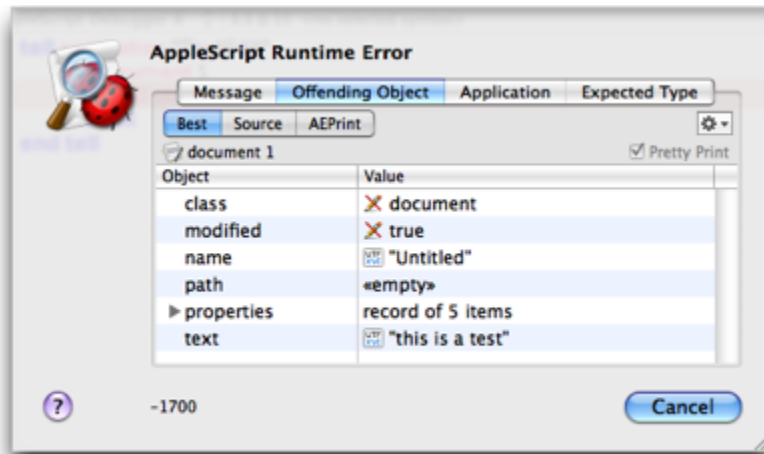
- Use the red mark in the scroll bar as a guide, or choose Edit > Go To Last Error (this immediately scrolls the window to bring the problematic line into view).

Compile Error

A compile error is reported as a text message from AppleScript. Script Debugger displays this text as a dialog.

Runtime Error

A runtime error can result in a more elaborate message from AppleScript. Therefore, the error dialog is more elaborate as well.



The message can contain up to six parts. Five of these correspond to the five parameters of the AppleScript error command, plus there is an Application parameter supplied by AppleScript. In the dialog, Script Debugger presents up to five of these parts of the message as individual panes, which you access through the buttons at the top of the dialog. The sixth part, the error number, is shown at the lower left of the dialog. Here is the correspondence between error command parameters and how they are shown in the dialog:

message string

Shown as the Message pane.

number

Shown at the lower left of the dialog.

partial result

Shown as the Partial Result pane.

from

Shown as the Offending Object pane.

to

Shown as the Expected Type pane.

[application]

Shown as the Application pane.

A pane of the error dialog might consist of a [viewer](#). If the value shown is an object reference, [Best view](#) is an [explorer view](#), and individual lines can be double-clicked to create a separate [viewer window](#).

If your script catches (handles) a runtime error with a try block, the error does not percolate up to AppleScript and it is not reported back to Script Debugger. In effect, there is no error. In [debug](#) mode, however, you can [break](#) on an error even if it is caught.

For information about a special kind of error situation where a file or transaction is left open, [read on](#).

Further Details:

[Leaks](#)

 [Variables](#)

[Apple Event Log](#) 



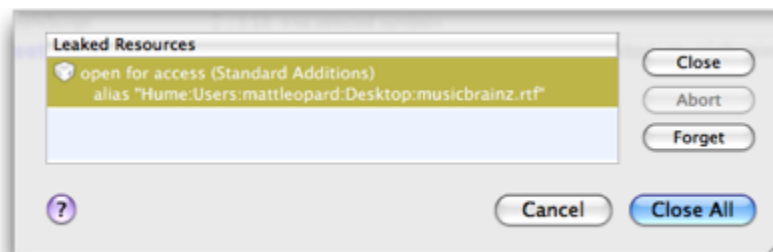
Leaks



If an uncaught runtime [error](#) occurs, or you [interrupt](#) execution, when your script still has certain links to the outside world in an open state, Script Debugger helps out by giving you the chance to close them. There are two primary situations:

- Your script has left a file open (using the `open for access` command).
- Your script is in the middle of a transaction (talking to FileMaker Pro, the only application that implements transactions) and has left the transaction open.

In a case like this, Script Debugger will show its Leaks dialog.



In the above illustration, the dialog shows a list of files that have been left open (there happens to be just one). Typically, you should click the Close All button. This closes all open files.

Alternatively, you could select a file and click the Forget button. This means to leave the file open (basically you're telling Script Debugger not to help out here), but that isn't something you're likely to want to do, since the leak then remains.

In the case of a transaction, you can close the transaction to "commit" it, or click Abort to "roll back" the transaction.

The Leaks dialog can be summoned any time open resources exist, by choosing Script > Show Leaks. (You can also add a Leaks button to the script window's [toolbar](#).)

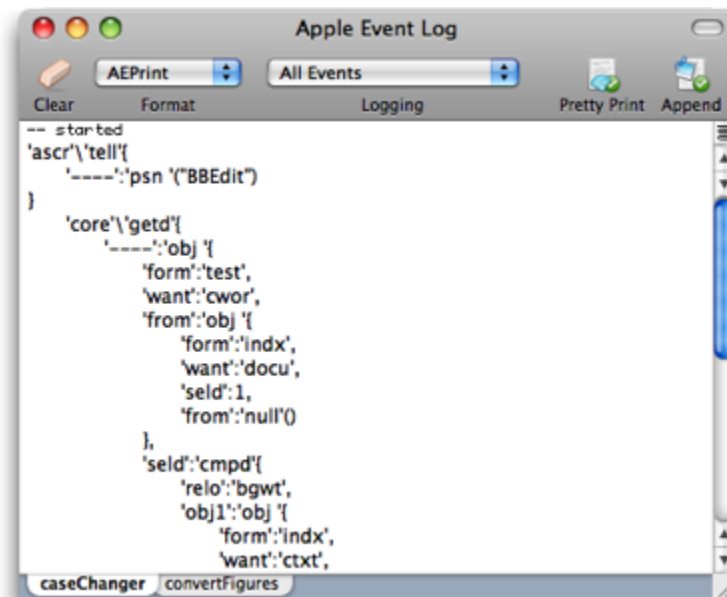


Apple Event Log



The Apple Event Log is a place where your script can generate messages during execution (with the AppleScript `log` command), plus you can keep track of the Apple events that pass between your script and target applications. To **summon the Apple Event Log window**:

- Choose Window > Apple Event Log. (No logging takes place unless the window is open.) You can also add an Apple Event Log button to the script window's [toolbar](#).



The tabs along the bottom of the Apple Event Log window list open script windows. There is only one Apple Event Log window, but you can have many scripts open, so each script is logged into its own tab.

You can determine whether (and how) each script performs logging when it executes. You can make this determination — and you can actually run the script — while working *either* in the [script window](#) or in the Apple Event Log window:

- If you're working **in a script window**, use the View menu to control logging, and use the Script menu or the toolbar to control script execution. As you switch between script windows, the Apple Event Log window automatically brings forward the tab of the frontmost script window, so you can watch the log easily.
- If you're working **in the Apple Event Log window**, use the toolbar or the View menu to control logging, and use the Script menu to control script execution. To determine which script you're controlling, use the tabs at the bottom of the Apple Event Log window. Whichever tab is frontmost, that's the script you're controlling. (You can also bring a particular script window to the front, by double-clicking its tab.)

The toolbar's Format popup menu items (and the corresponding menu items in the View menu) determine the "language" in which subsequent logging will be shown:

- **Source** (or View > Log As Source) means the compiled [OSA language](#) of the script being run. Typically, this will be AppleScript.
- **AEPrint** (or View > Log As AEPrint) means that the entire structure of Apple events is shown in [AEPrint format](#).
- **Raw** (or View > Log As Raw (Chevron) Events) is like a combination of the other two. Logging is done in the source language, but individual terms are shown as [raw Apple event codes](#), much as you can do with a [script](#) or [dictionary](#).

The toolbar's Logging popup menu items (and the corresponding menu items in the View menu) determine what will be logged:

- **Nothing** (or View > Log Nothing) means that logging is turned off for this script. (Logging slows down a script's execution, so if the Apple Event Log window is open, for maximum speed when you run a script, turn off logging for that script.)
- **Log Events** (or View > Log Log Events) means that only `log` commands in the script will cause the log to be written to. A `log` command in your script has no effect unless the Apple Event Log window is already open and the Logging setting is not Nothing. A `log` command causes the logged value to appear in the Apple Event Log window as if it were a multi-line comment, (`*thus*`).
- **All Events** (or View > Log All Events) means that `log` commands and all outgoing Apple events will be logged.
- **All Events & Replies** (or View > Log All Events & Replies) means that `log` commands and all outgoing Apple events and the corresponding replies from the target application(s) will be logged.

Even if all events are being logged, your script can temporarily disable logging of outgoing Apple events, in code, by issuing the `stop log` command. (And then logging can be enabled by issuing the `start log` command.) The `stop log` and `start log` commands are built into AppleScript, but only Script Debugger obeys them (you can't issue these commands in Apple's Script Editor).

You can control the Apple Event Log window [modes and views](#). Also, when you're [debugging](#), the Apple Event Log window takes on [additional powers](#).

Further Details:

[Logging Modes and Views](#)



Logging Modes and Views



You can determine how the [Apple Event Log window](#) displays material. These settings apply to the tab that is frontmost, not to the Apple Event Log window as a whole.

Logging is performed in one of two **append modes**. Either new logged material is appended to the material already present (append mode is on), or else the log is cleared automatically as the script begins to run (append mode is off). To determine the append mode:

- Choose View > Append To Log. If the menu item is checked, append mode is on. Alternatively, use the Append button in the Apple Event Log window [toolbar](#) to toggle the mode. If the button says “Append”, append mode is on.

To clear the log manually when the Apple Event Log window is frontmost, choose Edit > Delete (or click the Clear Log button in the [toolbar](#)).

The option to **pretty-print** logged material makes a difference when you’re viewing in [AEPrint](#) format, and also when lists and records are shown in Source format. Nested structures are more readily comprehended when [pretty-printing](#) is turned on. To determine the pretty-print setting, bring the Apple Event Log window to the front, and then:

- Choose View > Pretty Print. If the menu item is checked, pretty-printing is turned on. Alternatively, use the Pretty Print button in the [toolbar](#) to toggle the mode. If the button says “Pretty Print”, pretty-printing is turned on.

Other view settings that you can make when the Apple Event Log window is frontmost are parallel to those for a script window. You can turn [wrapping](#) on or off, and you can show [tab stops](#), [invisible characters](#), and [spaces](#).

All the options mentioned on this page can be set as your [defaults](#) for the Apple Event Log window tab belonging to a new empty [script window](#).



Record



You can record user actions in a recordable application. A recordable application (such as BBEdit, or the Finder) translates user actions into the AppleScript code that would be used to perform those same actions programmatically. This can be useful as a way of learning to script that application.

To **record user actions**:

- Choose Script > Record (or click the Record button in the toolbar) and switch to the recordable application. Alternatively, if you're already in the recordable application, you can choose Record from Script Debugger's dock menu.

Perform actions in the recordable application. Script Debugger will record the AppleScript equivalent of each action. When you're done recording, switch to Script Debugger and choose Script > Stop (or click the Stop button in the toolbar, or choose Stop from Script Debugger's dock menu).





Default Target



Script Debugger lets you **set an application as your script's implicit target** (the application to which undirected Apple events should be sent). To do so:

- Choose Script > Default Target and select the desired target application.

The application should be running or in the list of [known applications](#) (or it can be AppleScript Studio, which appears at the top of the list). When you run your script, the target application must be running. If it isn't, Script Debugger will offer to launch it.

This feature is useful for simulating runtime environments where there is in fact a significant implicit target. For instance, if you're going to run a script from BBEdit's Scripts menu, code targeting BBEdit doesn't have to appear in a tell block, because you're "inside" BBEdit already. So you could have a script that uses BBEdit commands and terminology with no tell block. To test or run such a script from within Script Debugger, you need a way to make BBEdit the default target, and that's what this feature provides.



Parent Script



Script Debugger lets you **set another script as the implicit parent** of the current script (using AppleScript's script object inheritance mechanism). Both scripts must be open. Then:

- Bring the "child" script frontmost and choose Script > Parent Script and the name of the parent script. (You can also add a Parent Script popup menu to the script window's [toolbar](#).)

This parent-child relationship persists only while both scripts are open. If you try to close the parent script, you'll get a dialog warning that you're about to break the parent-child relationship.

This feature is useful for testing individual handlers in a script without altering that script. It can be employed to make a set of top-level entities (properties, handlers, etc.) from one script available in another, or for simulating a runtime environment where such parent script relationships are used.



[Default Target](#)



Debug



Debugging a script is like [running](#) it, except that as it runs in [debug mode](#), **your script can pause in the middle**. Script Debugger can debug (and [run](#)) multiple scripts simultaneously.

This opens up all sorts of new possibilities. You can see (and alter!) the values of your script's [variables](#), and [other AppleScript values](#), as they change during the course of execution. You can see [what code is executed](#) and what [choices](#) your code makes. The [Apple Event Log window](#) also takes on new powers. Thus, with debugging, you can learn much more about how your script operates.

Debugging is a separate mode. A script is either in debug mode or it isn't. When you [enter debug mode](#), the window changes slightly, to accommodate things like [breakpoints](#) and [code coverage](#) in the [gutter](#), and additional menu items (and toolbar buttons) become active to permit [stepping](#) through the code.

As your code executes in debug mode, it can [pause](#), and thus the big question is, *when* will it pause? The answer involves chiefly the interplay between [breakpoints](#) and the [stepping](#) commands.

Further Details:

- [Turning On Debugging](#)
- [Pause](#)
- [Execute When Debugging](#)
- [Breakpoints](#)
- [Step](#)
- [Trace](#)
- [Call Stack](#)
- [Variables \(Debug Mode\)](#)
- [Expressions](#)
- [Exceptions](#)
- [Code Coverage](#)
- [Apple Event Log \(Debugging\)](#)
- [External Debugging](#)



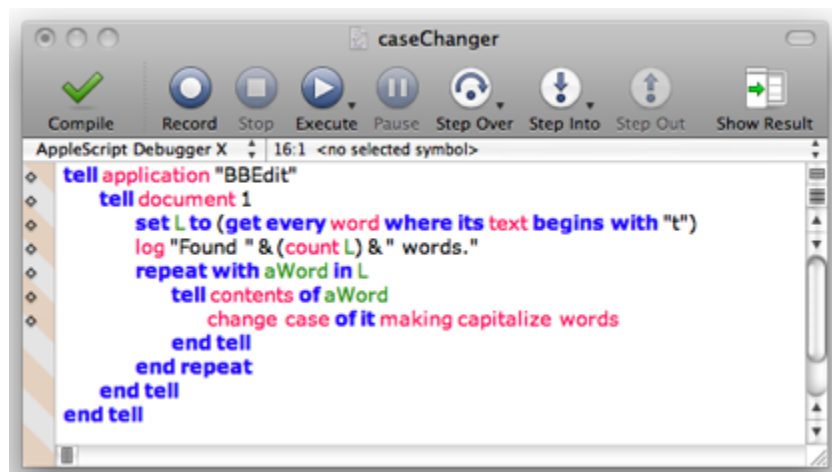
Turning On Debugging



Debugging is a separate mode. A script is either in debug mode or it isn't. To **turn on debug mode** for a script:

- Choose Script > Enable Debugging. If the menu item is checked, debugging mode is on.

The [script window](#) appearance changes slightly, as shown in the illustration below:



Notice the indications in the above illustration that you're in debug mode:

- The [Step Over and Step Into](#) buttons are enabled in the toolbar.
- The [language popup](#) has changed from AppleScript to AppleScript Debugger X.
- The [gutter](#) has widened and contains diamonds (to show where [breakpoints](#) can go).

To turn off debugging mode:

- Choose Script > Enable Debugging again, to uncheck it.

When you're finished debugging, you will probably want to save your script *not* in debugging mode. A script left in debug mode (with its language set at AppleScript Debugger X) is not portable to machines that don't have Script Debugger, and won't run normally in other environments. (Apple's Script Editor cannot even open such a script.) Having saved a script in debug mode, you would not want to distribute it to other users accidentally. (Also, a script in debug mode runs slower and uses more of AppleScript's internal resources.)

There is, however, one good reason for deliberately saving a script in debug mode and running it elsewhere — so that you can [debug externally](#).

[Pause](#) 



Pause



The difference between [debug mode](#) and normal mode is that in debug mode your script can **pause** while executing. The key to debug mode's behavior is the relationship between the various things that can cause your script to pause.

When a script is paused:

- The Execute command becomes the Resume command. The name changes in the Script menu and in the toolbar.
- In the Script menu, the Pause menu item is changed to Paused and is checked.
- In the toolbar and the Script menu, the [step](#) commands (and Stop) are enabled.
- In the [Window menu](#) (and the [Windows Inspector](#)), the listing for this script window is badged with an icon indicating that it is paused.
- The all-important blue arrow in the [gutter](#) indicates the line at which your script is paused; this line has *not* yet been executed.



Visualize what happens when your script runs. One line is executed, then another, then another. There are branches, so some code might not be executed. There are loops and handlers, so some code might be executed several times. Like a mouse running in a maze, the computer traces a **path of execution** through your code. (You can actually watch this happening in debug mode, by [tracing](#).)

- A [breakpoint](#) is a line where, if the path of execution comes to it, execution will automatically pause. It pauses *before* executing the breakpointed line. Breakpoints take priority over everything. When you are in debug mode, no matter how you cause execution to proceed, the script will pause when the path of execution hits a breakpoint (unless you have unchecked Script > Break on Breakpoints).
- The [step](#) commands cause execution to proceed by a limited amount. A step command means, "Start or resume executing, and pause when you come to a certain thing," where each step command has a different idea of what that certain thing is. But remember, breakpoints take priority. If the path of execution hits a breakpoint before hitting the thing the step command is looking for, execution will pause (unless you have unchecked Script > Break on Breakpoints).
- If you've elected to [break on exceptions](#), encountering a runtime error (even an error that your script catches and handles) will pause your script.
- During a lengthy bout of execution, you can **manually pause the script** by choosing Script > Pause (or use the Pause button on the toolbar).

Distinguish stopping from pausing!

- The Pause button, and all the other ways of pausing, leave you somewhere in the middle of execution. From here, you can proceed further, even completing the script normally if you want to.
- The Stop button (or choosing Script > Stop) aborts execution right where it is and returns everything to a completely neutral state. If you execute the script now, you'll be starting at the very beginning once more.

While you are paused, you can examine the state of your script. You can view the [call stack](#), the values of your [variables](#), the values of [expressions](#), and the [Apple Event Log window](#).

Then you can make your script [proceed](#) once again.

Tip: If your script is paused but you don't know where (because you've scrolled to examine some other region of the script), choose Search > Go to Current Line to bring the line containing the blue arrow into view.



[Turning On Debugging](#)

[Execute When Debugging](#)





Execute When Debugging



Here is a summary of the ways you can **start or resume execution** when you're in [debug mode](#).

- Choose Script > Execute or Script > Resume (they are the same menu item), or use the Execute or Resume button in the toolbar (they are the same button).
- Choose Script > Trace, or Option-click the Execute or Resume button in the toolbar. Execution will proceed in [trace mode](#), which is slower than normal execution so that you can see the [path of execution](#) as it occurs.
- Issue any [Step command](#).
- Issue the [Execute \(or Trace\) to Here](#) command. This is a way of setting a breakpoint and resuming execution at the same time.



Breakpoints



A **breakpoint** designates an executable line of code where your script will [pause](#) if the path of execution reaches it. When a script pauses at a breakpoint, it pauses *before* executing the breakpointed line.

You can create a breakpoint only in [debug mode](#), but the breakpoint is not lost if you leave debug mode — it will still be there the next time you switch to debug mode. Breakpoints are [saved](#) when you save a compiled script in debug mode. They are lost when you save a compiled script in normal mode and close the script.

Places where you can set a breakpoint are shown with diamonds in the [gutter](#) of the script window.

To **set a breakpoint**:

- Select within the desired line, then choose Script > Set Breakpoint. Alternatively, click in an empty diamond. The diamond is filled with red, showing that a breakpoint has been set.

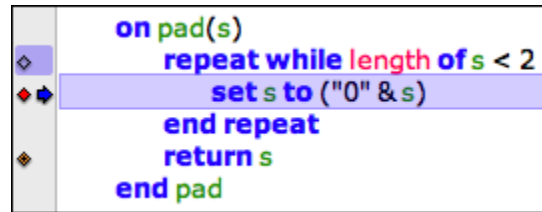
To **clear (remove) a breakpoint**:

- Select within the desired line, then choose Script > Clear Breakpoint. Alternatively, click in a red diamond. The diamond becomes empty, showing that there is no breakpoint there. You can clear *all* breakpoints at once by choosing Script > Clear All Breakpoints.

You can also leave all breakpoints in place but **turn them off temporarily** by choosing Script > Break on Breakpoints. If the menu item is unchecked, breakpoints are disabled. Encountering a breakpoint in the [path of execution](#) will *not* cause your script to pause. (If breakpoints appear to have mysteriously stopped working, check to make sure that Break on Breakpoints has not gotten unchecked!) Also, there is an optional [toolbar](#) icon you can add to the script window toolbar; the icon toggles its state to show whether we're going to break on breakpoints or not.



It is also possible to **make a temporary breakpoint**. A temporary breakpoint is a breakpoint which, when it is encountered, clears itself. Thus, we will pause there, but only once (because after that, the breakpoint will be gone). This is convenient, for example, to pause the first time through a loop but not on subsequent iterations. To set a temporary breakpoint, choose Script > Set Temporary Breakpoint. Alternatively, Option-click in an empty diamond. The diamond gets a little + sign in it.



In the above illustration:

- The `repeat` line has an empty diamond. A breakpoint could go there, but there isn't one now. (The blue shading in the gutter indicates that execution has [passed through](#) that line.)
- The `set` line has a breakpoint (the diamond is red), and we are paused there (the blue arrow is there), *before* executing the line.
- The `return` line has a temporary breakpoint (there is a **+** sign in its diamond). When we proceed and reach it, we will pause there and the breakpoint will be cleared.

The [execute to here](#) facility is a convenient way to combine setting a breakpoint with [resuming](#) execution.

Further Details:

[Execute to Here](#)



Execute to Here



“Execute to here” means to start or resume execution until a certain line is reached, pausing at that line. You could do this by setting a breakpoint on the target line and executing (or resuming execution). In fact, the “execute to here” facility is a shortcut for doing exactly that.

To **use “execute to here”**:

- Select within the line that you want to execute to, and choose Script > Execute To Here. Alternatively, Shift-click in the empty diamond of the line that you want to execute to (the target line). A [temporary breakpoint](#) will be set there, and execution will [start or resume](#).

If a [breakpoint](#) is encountered before the target line is reached, we will pause at that breakpoint.

A variant of “execute to here” is “trace to here”, which is the same except that we [trace](#) instead of executing at normal speed. To **use “trace to here”**:

- Choose Script > Trace to here. Alternatively, Option-Shift-click in the empty diamond of the target line.



Step



There are three **Step commands** — Step Over, Step Into, and Step Out. You can choose them from the Script menu or click the buttons in the toolbar. We'll use the code in the illustration below to show what they mean. In the illustration, we are paused at a breakpoint at line 9 (without yet having executed line 9).

```
1  script s
2    property chapNum : 3
3    property counter : 2
4    on pad(s)
5      repeat while length of what < 2
6        set what to ("0" & what)
7      end repeat
8    end pad
9    set x to pad(chapNum as string) & pad(counter as string)
10   return x
11 end script
12 set res to run s
13 return "fig" & res
```

- **Step Into** is the simplest. It means, "Execute the current line of code, and then, wherever the path of execution takes you, pause right there, on the next line that would be executed."

So, in the illustration above, Step Into would cause the script to pause at line 5. Why? Because line 9, where we are paused, calls the pad handler. So when we execute it, we'll dive into the pad handler, and the next executable line where we can pause, in that path of execution, is line 5.

- **Step Over** is similar to Step Into, except that it follows an additional rule, "Don't pause in a deeper level of the call stack than where you are right now."

So, in the illustration above, Step Over would cause the script to pause at line 10. Why? Because that's the next executable line that isn't at a deeper level. Line 9, where we are paused, calls the pad handler, which is a deeper level, so we don't pause until the next executable line *after* the path of execution has returned from the pad handler.

- **Step Out** means, "Execute until you come to the next executable line at a higher level of the call stack than where you are right now, and then pause."

So, in the illustration above, Step Out would cause the script to pause at line 13. Why? We are paused at line 9, in s's implicit run handler. We execute to the end of the run handler, which is line 10, and return from s's implicit run handler. Now we are at a high level, so we want to pause. In fact, we are in line 12, because that is where s's run handler was called. But we don't pause in line 12, because if we were going to pause there, it would be *before* executing line 12 and *before* telling s to run. So now we're at line 13.

All of those details are predicated on the supposition that no [breakpoints](#) are encountered. Suppose, for example, that there were a breakpoint at line 5 (and assume that Script > Break on Breakpoints is checked). The path of execution passes through line 5, so all three commands would do exactly the same thing — pause at line 5. Breakpoints [take priority](#) over everything!

Both Step Over and Step Into can be used not only to resume but also to start execution of a script. In this case they both pause before the first executable line of the script.

Both Step Over and Step Into have the same [options for executing handlers](#) as the Execute button.



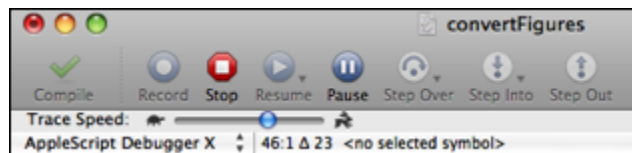
Trace



Tracing is like [executing](#), only it's slower — slow enough that you can actually see the blue arrow moving along the [path of execution](#). **To trace:**

- Choose Script > Trace (or Option-click the Execute / Resume button in the toolbar).

While tracing, a speed slider appears below the title bar and toolbar of the script window. You can adjust the tracing speed (from “tortoise” at the left to “hare” at the right) while tracing is going on.



Tracing is in one sense just a slower form of execution, and will [pause](#) for the same reasons (e.g., because a [breakpoint](#) is encountered, or because you issue the [Pause](#) command). However, what's really happening is that Script Debugger is pausing and resuming after every executed line. This means that you can see more than the blue arrow moving — you can also see the [call stack](#) growing and shrinking, and the [variable values](#) changing, just as you would if you were repeatedly issuing the [Step Into](#) command.

Trace has the same [options for executing handlers](#) as Execute.

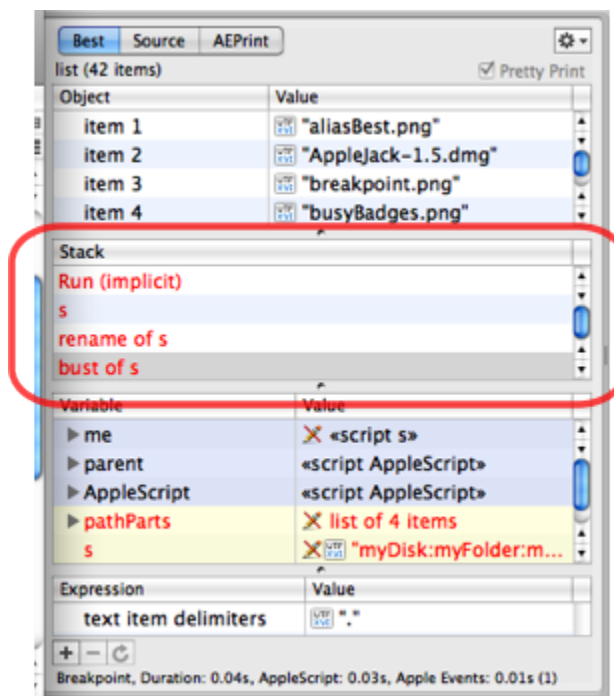


Call Stack



The **call stack** is the chain of handler calls currently executing in your script. Execution typically starts with the script's run handler (implicit or explicit). Code in this handler can call another handler, which can call another handler (or the same handler, recursively), and so on, so that at any given moment during the execution of your code, there is a nest or chain of handlers leading down from top level to the line currently being executed.

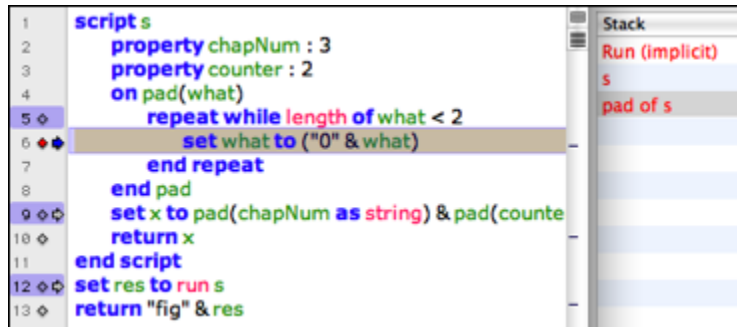
The call stack is shown in the second pane of the [result drawer](#) (the word "Stack" appears at the top).



The stack grows downward. In the illustration above, the script's implicit run handler told the script object `s` to run, `s`'s implicit run handler called the `rename` handler, and the `rename` handler called the `bust` handler. Each entry in the stack is called a *stack frame*.

Stack frames newly added since the last time the script paused are shown in red. In the illustration above, *all* the stack frames are in red because this is the first pause since the script started executing.

Reflecting the call stack back in the script's text area is a little tricky, because as long as there are multiple stack frames, we can be paused at more than one place simultaneously. You can see this happening in this illustration:



The implicit run handler of the main script (line 12) has called `s`'s implicit run handler, which has called `pad` (line 9), and we are now paused inside the `pad` handler (line 6). Thus there is a sense in which we are paused at line 6 and line 9 and line 12 simultaneously. To indicate this, line 6 (the next line we'll actually execute) has a blue arrow, and the others have a hollow (white) arrow. The blue arrow shows the current line in the *current stack frame*.

You can select a line of the call stack pane to explore different stack frames. When you do, three things will happen:

- Back in the script's text area, the line where we're paused in the selected stack frame is highlighted. So, in the above illustration, if you click the `s` line in the call stack pane, then back in the script's text area, line 9 is highlighted.
- The [variables pane](#) changes to reflect the variable values in scope in the selected stack frame.
- [Expressions](#) are re-evaluated against the selected stack frame.



Variables (Debug Mode)



During debugging, the [variables pane](#) in the [result drawer](#) plays an expanded role. Each time the script pauses, the names and values of *all* variables currently in scope are displayed. Thus, the variables pane is an important tool for understanding what your script is doing. In the variables pane:

- Variables inherited from the higher AppleScript world come first, and have a blue background.
- Variables local to the current scope come next, and have a yellow background.
- Variables globally visible to the current scope come last, and have a white background.
- A variable whose value has changed since the last pause is displayed in red.

Variable	Value
me	«script s»
parent	«script AppleScript»
AppleScript	«script AppleScript»
bothLists	list of 2 items
extension	"tiff"
n1	"3"
n2	"2"
newFileName	"as_0302.tiff"
oldPath	"myDisk:myFolder:myFolder2:myPicture.tiff"
pathPart	list of 3 items
item 1	"myDisk"
item 2	"myFolder"
item 3	"myFolder2"
s	«script s»
chapNum	3
counter	2
thisFile	"myDisk:myFolder:myFolder2:myPicture.tiff"


What variables are shown depends not only upon the current scope but also on the current [stack frame](#). Click on a line of the call stack, and the variables listed will change to reflect the selected stack frame.

There is usually a result each time script pauses, namely the result of the most recently executed line of code. This is shown in the [result pane](#), not the variables pane.

Important note: To have local variables appear during debugging, you should explicitly declare them (with a `local` statement) in your script. This is especially the case with local variables in handlers.

Another way to see variable values is through tooltips that appear when you hover the mouse over an expression in your script. A [Debugger preference](#) sets the conditions under which these appear.

The variables pane is an [explorer view](#), with [all the abilities](#) that this entails. For example, individual lines of the variables pane can be separated into individual [viewer windows](#). If you leave open a viewer window on a variable that you're interested in, it will be updated automatically each time the script pauses.

Furthermore, variables without a "read-only" icon () are [editable](#) in the variables pane! Select a line and press the Return key (or choose Edit Value from the contextual menu). Using this feature, you can experiment with the behavior of your script as it runs.



[Call Stack](#)

[Expressions](#)



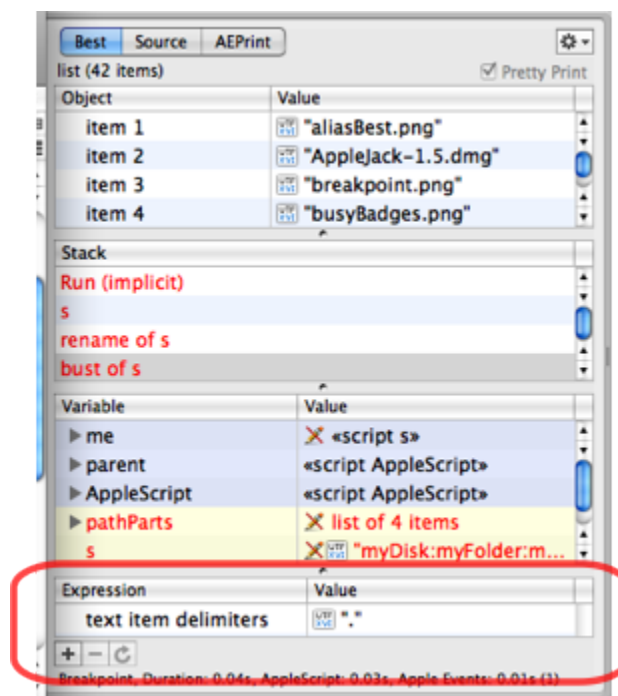


Expressions



An **expression** is a piece of AppleScript code that has a value. Expressions are evaluated every time the script pauses.

You create expressions and read their values in the expressions pane, the fourth pane in the [results drawer](#). You do not have to be in [debug mode](#) to create an expression. Expressions are [saved](#) when you save the compiled script.



- Use the **+** button to **create an expression**, or you can choose Script > New Expression.
- A convenient shortcut is Script > Copy To Expressions, which copies the **currently selected text** in your script as a new expression.
- To **remove an expression**, select it and use the **-** button, or press the Delete key.
- To **remove all expressions**, choose Script > Clear All Expressions.
- The circular arrow button forces the currently selected expression to be **re-evaluated**.

Expressions have certain similarities to the [variables pane](#):

- The expressions pane is an [explorer view](#), and an entry in it can be opened as a separate [viewer window](#), whose value is automatically updated at each pause.
- An expression whose value has changed since the last pause is shown in red.
- An expression is re-evaluated when you select a different stack frame in the [call stack](#).

If an expression uses a name that is undefined at the point where we are currently paused and in the call stack context that is currently selected, its value is marked as undefined.

Note: Evaluating an expression is like running a little one-line script, and even a little one-line script can do powerful things. An expression that changes a variable's value, or calls a handler in your script, can be a valid expression and therefore can have side-effects each time it is evaluated.

◀ [Variables \(Debug Mode\)](#)

[Exceptions](#) ▶



Exceptions



An exception is a runtime error, and you can determine whether a runtime error should cause a pause when [debugging](#), like a breakpoint.

Distinguish between a handled runtime error and an unhandled runtime error. A handled runtime error is an error that occurs within a try block. It would not normally cause any break in the action, because AppleScript will just call the on error clause if there is one, or just abandon the try block and continue execution after the block. An unhandled runtime error, on the other hand, causes execution to abort entirely, so there is no issue about pausing there — the script will do more than pause, it will stop.

To pause at handled runtime errors:

- Choose Script > Break on Exceptions. If the menu item is checked, we will pause handled runtime errors. Alternatively, there is also an optional [toolbar](#) icon you can add to the script window toolbar; the icon toggles its state to show whether we're going to break on exceptions or not.



If Break on Exceptions is checked, a handled runtime error causes a pause when encountered in [debug mode](#). Execution pauses at the line where the runtime error was encountered, before the error is thrown, and variables and expressions are evaluated at that point.

To learn what the error was:

- Look at the bottom of the result pane. The error message is displayed there.



- Choose Script > Show Last Error, or click the red arrow at the point where the error occurred. This summons the normal [error dialog](#). (You can also add a Show Last Error button to the script window's [toolbar](#).)

If Break on Exceptions is unchecked, then a handled runtime error does not cause any pause. However, you can *still* learn, during any pause *after* the error is encountered (but before some other runtime error is encountered), *where* the error was, by choosing Search > Go to Last Error, and *what* the error was, by choosing Script > Show Last Error.



Code Coverage



Script Debugger can mark the **lines of your script that were actually executed**. This can help you survey the path of execution without [tracing](#) or [stepping](#). For instance, you can easily see whether there are areas of the script that are never being executed. To turn this feature on or off:

- Choose Script > Show Code Coverage. If the menu item is checked, code coverage is on, and lines subsequently encountered by the path of execution will be marked in blue in the [gutter](#) (and a small line appears in the vertical scroll bar).

```
6   on pad(s)
7   repeat while length of s < 2
8     set s to ("0" & s)
9   end repeat
10  return s
11  end pad
12
13  on bust(s)
14    local pathParts, nameParts
15    set text item delimiters to ":"
16    set pathParts to text items of s
17    set text item delimiters to "."
18    set nameParts to text items of last item of pathParts
19    return {pathParts, nameParts}
20  end bust
21
22  on rename(n1, n2, oldPath)
23    local bothLists, extension, pathPart, newFileName
24    set bothLists to bust(oldPath)
25    set extension to last item of item 2 of bothLists
26    set pathPart to items 1 thru -2 of item 1 of bothLists
27    set newFileName to "as_" & pad(n1) & pad(n2)
28    set newFileName to newFileName & "." & extension
29    set text item delimiters to ":"
30    return (pathPart as string) & ":" & newFileName
31  end rename
```

To clear code coverage marks without turning code coverage off:

- Choose Script > Clear Code Coverage.

Code coverage marks are also removed when you start to execute the script.



Apple Event Log (Debugging)



During debugging, the [Apple Event Log window](#) takes on an extra power. The Go To Source menu item in the window's contextual menu becomes active. This menu item allows a linkage between an **entry in the log** and the **line in the script** that generated it:

- Control-click on a log entry and choose Go To Source. The script window comes to the front with the corresponding line highlighted.

Remember also that you can control execution of your script without leaving the Apple Event log window. Menu items (as well as buttons that you can optionally add to the Apple Event Log window's [toolbar](#)) let you issue the [Execute](#) and [Step](#) commands while the Apple Event Log is frontmost.



External Debugging



External debugging is a mechanism that lets you run a script elsewhere but summon Script Debugger to debug the script anyway. To use it:

- Save a compiled script while the script is in [debug mode](#). Now trigger that script in some other application. The script opens in Script Debugger, [optionally](#) paused before the first executable line. Now you can proceed to debug the script in the normal way.

It makes a difference whether the script is open in Script Debugger when it is triggered. If it is, then when the script finishes executing in debug mode, it remains open. If it isn't, then when the script finishes, it will close.

If the script was *not* open in Script Debugger beforehand, the name of the script, in the script window's title bar, will appear as the name of the host application with "(Debugging)" appended to it, as a sign that external debugging is proceeding in a temporary window.

It is often easier to use external debugging and test a script under the conditions in which it will actually run than to try to simulate those conditions artificially. A good candidate for external debugging is an applet, a folder action, an Apple Mail rule script, a BBEdit menu item script, or any script that is to be triggered automatically by some other application.

External debugging is especially useful when parameters are supplied as part of the call that triggers the script, since it shows you what those parameters are. Moreover, the handler and parameters are remembered as part of the [call history](#). In subsequent testing, therefore, you can call the same handler again, yourself, with the same parameters.

Here's an example, using an Apple Mail rule script. A rule script is structured like this:

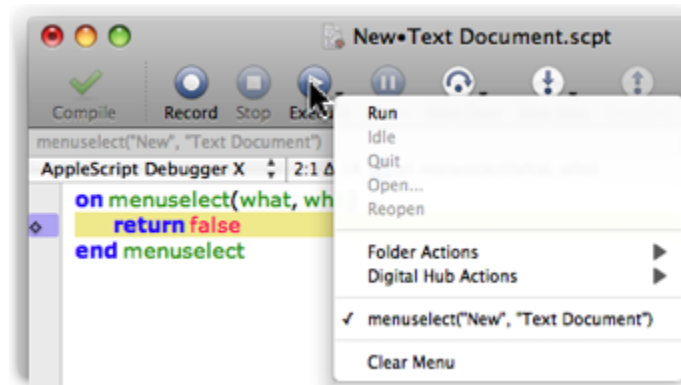
```
using terms from application "Mail"
    on perform mail action with messages theMessages for rule theRule
        tell application "Mail"
            -- do useful things here
        end tell
    end perform mail action with messages
end using terms from
```

You can compile that, put it in debug mode, place a breakpoint on the "tell" line, save it, and then (in Mail) nominate this script as the Run AppleScript action in a Rule. When the rule is triggered (which you can do by selecting some messages and choosing Message > Apply Rules), Script Debugger comes to the front, paused at the breakpointed line. Now you can examine theMessages to see what messages Mail has matched up with your rule's criteria. Moreover, the call to perform mail action has been added, with these parameters, to the event history. You can choose it to make it the current event and run the event handler repeatedly with the same parameters without switching to Mail.

Here's another example, using BBEdit's menu item script mechanism. A BBEdit menu item script is structured like this:

```
on menuselect(theMenu, theItem)
    return false
end menuselect
```

Save that, in debug mode, with the name *New•Text Document*, in *~/Library/Application Support/BBEdit/Menu Scripts*. Leave it open. Now switch to BBEdit and choose File > New > Text Document. This triggers our script; BBEdit calls our `menuselect()` handler. And the call to `menuselect()`, with the actual parameters, is added to the event history. You can thus repeat the call, with those same parameters, without switching to BBEdit.



◀ [Apple Event Log \(Debugging\)](#)



Reference



This is the reference section.

- It surveys Script Debugger's [menus](#).
- It describes Script Debugger's [preferences](#).
- It discusses Script Debugger's [windows](#) (in particular, the [inspectors](#)).
- It provides a brief [glossary](#) and answers some [frequently asked questions](#).

Further Details:

[Menus](#)
[Preferences](#)
[Windows](#)
[Glossary](#)
[Frequently Asked Questions](#)



[Develop](#)

Menus

This is a reference section describing all of Script Debugger's menus.

You can **customize the keyboard shortcut** for a menu item:

- Open Script Debugger's Preferences window and go to the [Key Bindings pane](#).

Hint: In some cases where you find yourself using a command often, you might be able to manage without a keyboard shortcut, because a [toolbar](#) item is supplied. Choose View > Customize Toolbar and examine the available toolbar items. If the command has a toolbar item, you can drag it into the toolbar, and from then on you can use that toolbar item as a way of issuing the command. This may prove sufficiently convenient.

Further Details:

[Application Menu](#)
[File Menu](#)
[Edit Menu](#)
[Search Menu](#)
[Script Menu](#)
[Dictionary Menu](#)
[View Menu](#)
[Window Menu](#)
[Clippings Menu](#)
[Scripts Menu](#)
[Help Menu](#)



Application Menu



For **Send Us Email** and **Check For Updates**, see the [Help menu](#).

For **Preferences**, see the [preferences](#) section.



File Menu

New Script. Creates a new [script window](#) according to your [saved defaults](#), if any.

New Script (No Defaults). Creates a new [script window](#) *ignoring* your [saved defaults](#). By default, this menu item is shown as an alternative to New Script when you hold down the Option key.

Open... Brings up a file dialog where you can choose a [script file](#) (to open it for [editing](#)) or an application (to open its [dictionary](#)).

Open Recent. Lists, and lets you open, recently opened scripts.

Open Selection in Viewer or **Open XXX Dictionary.**

- **Open Selection in Viewer** is the same as double-clicking a line in an [explorer view](#). It creates a separate [viewer window](#) for the selected line.
- **Open XXX Dictionary** is context-sensitive, based on the insertion point in your script window. It opens the dictionary for the application targeted by the [current tell block](#).

Open Dictionary. Presents a hierarchical menu where you can [open the dictionary](#) of installed scripting additions, AppleScript, AppleScript Studio, running scriptable applications, and applications in the [known applications list](#). To open the dictionary of an application not listed here, choose Application (the second item in this menu).

Recover Script. Recovers the [text](#) of a compiled script file, if possible.

Close, Close All. Attempts to close the frontmost window, or all windows. If a script window is “dirty”, you’ll be offered a chance to save it (or to decline to close it).

Save. Saves the frontmost script. If the frontmost script has never been saved, works like Save As.

Save As... Brings up a file save dialog, for saving the frontmost script as a new file, possibly in a different [format](#).

Save A Copy As... Like Save As, except that afterwards the script window shows the old file, not the newly created file.

Save All. Performs a Save on every “dirty” script window.

Revert To Saved. Opens the frontmost script from its previously saved state, throwing away any changes made since the last save.

Export. Saves a copy of the script as:

- a [run-only script](#), or
- as a [flattened script](#) that incorporates the script’s libraries

Reveal in Finder. Reveals the current document or selection in the Finder:

- If the current document is a saved script file, reveals that file in the Finder.
- If the current window is an application's dictionary, reveals that application in the Finder.
- If the current selection is in an [explorer view](#) and the selected item is a reference to a Finder item (a file or folder), reveals that item in the Finder.

Edit With BBEdit. Actually, might say Edit With TextWrangler or Edit With TextMate instead; it depends which of these applications you have, and which of them is running. Opens the current script document in the target application, initiating an [external editing session](#).

Description... Brings up a dialog for editing the script's [description](#).

Libraries... Brings up a dialog for editing the script's [libraries](#).

Manifest... Brings up a dialog displaying the script's [manifest](#), the applications and scripting additions on which it depends.

Script Format. Lets you set the [format](#) in which the script should be saved. Only formats compatible with the script's present format are enabled; to change to another format, choose File > Save As.

Application Options. If the script is an [applet](#) (script application), lets you set its applet options.

Page Setup..., Print... Brings up the usual dialogs for printing the frontmost script or dictionary. You can force a page break in your printed output by including the word !pagebreak! in a comment in your script (in the output, a page break will be substituted for this word).

Edit Menu

Undo, Redo. Moves the state of the frontmost script back or forward through the undo list.

Cut, Copy, Paste. The usual commands for moving material on and off the clipboard. By default, Copy Value appears when you hold down the Shift key (enabled if the selection is a line of an [explorer view](#)).

Paste As String Literal. Pastes the contents of the clipboard, with internal quotation marks, tabs, and line-end characters escaped, and wrapped in quotation marks (unless the insertion point is already inside a string literal).

Paste Tell. Inserts a [tell block](#) for a running scriptable application or a [known application](#). If no script window is open, or if you hold down the Option key, creates a new script window containing the tell block.

Delete. Clears the selected text without moving it to the clipboard. Also works on any selectable removable entity (an [expression](#), a [library](#), etc.). Clears the contents of the frontmost tab of the [Apple Event Log window](#) if the log window is frontmost.

Complete. Presents a list of possible [completions](#) (or the only known completion) for the AppleScript term that starts to the left of the insertion point. Alternatively, press Esc.

Select All. Selects all text in the current selection context.

Balance. Selects [surrounding delimiters](#) or [block boundaries](#), or beeps.

Split Editor Vertically, Split Editor Horizontally, Close Split View, Close All Split Views. Manipulates [splitting](#) of the [script window](#) editing area. By default, Close All Split Views appears when you hold down the Option key.

Shift Left, Shift Right. Removes or adds a [level of indentation](#) to the lines containing the selection.

Comment, Uncomment. Adds or removes a [level of comment characters](#) to the start of the lines containing the selection.

Entab, Detab. Changes leading indentation to [tabs or spaces](#) in the lines containing the selection.

Spelling. Accesses the built-in Mac OS X spell-checking mechanism.

Special Characters... Brings up the system's Character Palette for entering Unicode characters.

Search Menu

Find... Brings up the [Find dialog](#).

Find Again. Finds the contents of the Search For field in the Find dialog, forwards, starting at the current selection.

Find Again Backwards. Finds the contents of the Search For field in the Find dialog, backwards, starting at the current selection.

Find Selection. Enters the current selection into the Search For field in the Find dialog, and finds it, forwards, starting at the current selection.

Find Selection Backwards. Enters the current selection into the Search For field in the Find dialog, and finds it, backwards, starting at the current selection.

Enter Search String. Enters the current selection into the Search For field in the Find dialog.

Enter Replace String. Enters the current selection into the Replace With field in the Find dialog.

Replace. Replaces the current selection with the contents of the Replace With field in the Find dialog.

Replace & Find Again. Performs a Replace followed by a Find Again.

Replace & Find Again Backwards. Performs a Replace followed by a Find Again Backwards.

Replace All. Replaces all instances of the contents of the Search For field in the Find dialog with the contents of the Replace With field.

Look Up Definition. Copies the current selection into the search field of the [Look Up Definition inspector](#) and [performs the search](#).

Go to Line.... Brings up a dialog allowing you to [jump to a line](#) by its number.

Go to Current Line. Scrolls to the line that has the blue arrow in [debug mode](#).

Go to Last Error. Scrolls to the line that has the red arrow or stop sign icon, where the last error occurred.

Go to Next Handler, Go to Previous Handler. Selects in the first line of the handler definition following or preceding the current selection.



Script Menu



Compile. [Compiles](#) the script. To force recompilation even when Script Debugger thinks the script doesn't need compiling, choose **Recompile** (by default, this appears when you hold down the Option key).

Record. Turns on AppleScript [recording mode](#), so that user actions in recordable applications are written into the script.

Execute. [Runs](#) the script, compiling it first if necessary. Submenus allow certain [handlers](#) to be called individually. When [paused](#) in [debug mode](#), becomes **Resume**, and [continues execution](#) from the paused line.

Trace. In [debug mode](#), starts [tracing](#). Submenus allow certain standard [handlers](#) to be called individually.

Stop. Aborts the running script.

Pause. In [debug mode](#), [pauses](#) the script after the line currently being executed.

Step Over, Step Into, Step Out. The [step commands](#), used in [debug mode](#). Submenus allow certain standard [handlers](#) to be called individually.

Enable Debugging. Toggles on or off [debug mode](#).

Show/Hide Result. Opens or closes the [result drawer](#).

Show Result in Viewer. Shows the [result pane](#) as a separate [viewer window](#).

Show Last Error... Shows the [error dialog](#) for the most recently encountered error.

Show Leaks... Shows the [leaks](#) dialog.

Show Code Coverage. Toggles [code coverage](#) on or off.

Clear Code Coverage. Removes [code coverage](#) marks without toggling code coverage off.

Break on Exceptions. In [debug mode](#), toggles whether or not [runtime errors](#) cause a pause.

Break on Breakpoints. In [debug mode](#), toggles whether or not [breakpoints](#) operate at all.

Set/Clear Breakpoint. Creates or removes a [breakpoint](#) at the currently selected line.

Set Temporary Breakpoint. Creates a [temporary breakpoint](#) at the currently selected line.

Execute to Here. Sets a temporary breakpoint at the currently selected line and [starts or resumes execution](#).

Trace to Here. Sets a temporary breakpoint at the currently selected line and [starts tracing](#).

Clear All Breakpoints. Removes all [breakpoints](#).

New Expression. Creates a new empty [expression](#), ready for editing.

Copy To Expressions. Creates a new [expression](#) by copying the current selection.

Clear All Expressions. Deletes every [expression](#).

Default Target. Sets the [implicit target](#) for the script.

Parent Script. Sets the [parent of the script](#) to another currently open script.

 [Search Menu](#)

[Dictionary Menu](#) 



Dictionary Menu



Reload. In the current [explorer view](#), reloads information that's hierarchically dependent on the currently selected line. To reload *all* information in the explorer, choose **Reload All** (by default, this appears when you hold down the Option key).

Paste Tell. Inserts into the frontmost script a [tell block](#) targeting the application of the current [dictionary](#) or [explorer](#). Hold down the Option key to insert this tell block into a new script window (**Paste Tell (In New Document)**). If what's being displayed in a dictionary window is a command, the tell block includes a template for issuing that command. If what's being displayed is an event, an event handler for that event is inserted instead of a tell block. If this is a dictionary explorer, the tell block includes a reference to the currently selected property or element (or element collection).

Launch/Activate XXX. Starts up or brings to the front the application XXX, which is the application of the current [dictionary](#) or [explorer](#).

Quit XXX. Quits the application XXX, which is the application of the current [dictionary](#) or [explorer](#).

Open in New Window. Opens a second window on the current [dictionary](#) or [explorer](#).

Back, Forward. Changes the content of the [info pane](#) of the current [dictionary](#), moving [back and forward](#) through previously viewed content.

Dictionary View, Explorer View. Switches the current dictionary/explorer window between [dictionary](#) and [explorer](#).

Show/Hide Diagram. Shows or hides the current dictionary window's [diagram drawer](#).

Show Inherited Properties. Toggles whether or not the [inherited properties](#) appear in this dictionary's display.

Show Inherited Elements. Toggles whether or not the [inherited elements](#) appear in this dictionary's display.

Show Extra Documentation. Toggles whether or not [extended explanatory content](#) appears in this dictionary's display.

Larger Text, Smaller Text. Increases or decreases the [size](#) of information in this dictionary's display.



View Menu

Show/Hide Toolbar. Toggles the display of the [toolbar](#) for the current window.

Customize Toolbar.... Brings up the dialog for customizing the contents of the [toolbar](#) for the current window type.

Show/Hide Navigation Bar. Toggles the display of the [language popup](#) and [navigation bar](#) in the current script window.

Best View, Source View, AEPrint View. Switches between [views](#) in the current viewer pane or viewer window.

Log As Source, Log As AEPrint, Log As Raw (Chevron) Events. Switches between [logging formats](#) for subsequently logged events in the frontmost tab of the Apple Event Log window.

Log Nothing, Log Log Events, Log All Events, Log All Events & Replies. Switches between settings for [what to log](#) in the frontmost tab of the Apple Event Log window.

Append To Log. Toggles whether or not the Apple Event Log window is [automatically erased](#) each time the script starts over from the beginning.

Show Line Numbers. Toggles visibility of [line numbers](#) in the current script window.

Show Tab Stops. Toggles visibility of [tab stops](#) in the current [script window](#), [viewer pane or window](#), or [Apple Event Log window](#) tab.

Show Invisibles. Toggles visibility of [invisible characters](#) in the current [script window](#), [viewer pane or window](#), or [Apple Event Log window](#) tab.

Show Spaces. Toggles visibility of [space characters](#) in the current [script window](#), [viewer pane or window](#), or [Apple Event Log window](#) tab.

Wrap Lines. Toggles [line wrapping](#) in the current [script window](#), [viewer pane or window](#), or [Apple Event Log window](#) tab.

Pretty Print. Toggles [pretty-printing](#) in the current [viewer pane or window](#), or [Apple Event Log window](#) tab.

Show Raw (Chevron) Syntax. Toggles whether or not terminology is shown as [raw Apple event codes](#) in the current [script window](#) or [dictionary window](#).



Window Menu



Inspectors. Sets [inspector visibility](#).

Minimize Window. Standard Mac OS X window minimization command.

Zoom Window. Standard Mac OS X window zoom command.

Bring All Windows to Front. Standard Mac OS X command for unlacing the application's windows from between those of other applications.

Set Default Script / Viewer Size & State. Saves the current [script window](#) or [viewer window](#) as a model for future new script windows or viewer windows, respectively.

Reset Default Script / Viewer Size & State. Reverts to Late Night Software's default style for future new [script windows](#) or [viewer windows](#).

Apple Event Log. Summons the [Apple Event Log window](#).

The Window menu also lists all open windows. Icons describe the state of each window (paused, unsaved, what application a dictionary shows, and so forth). The hierarchical arrangement of these menu items shows dependencies (such as a viewer window generated from a certain explorer or script). Tooltips show the relevant file's path.



[View Menu](#)


[Clippings Menu](#)





Clippings Menu



The Clippings menu () accesses [clippings](#). Each menu item represents a file (or folder) in the Clippings folder.

A file will appear as a menu item. A folder will appear as a hierarchical menu, and the files inside it will be its menu items. The name of a file (or folder) is the name that will appear in the menu, except that certain names or part-names are hidden and used for determining the order of the menu, as follows:

- If a name starts with the prefix `##` , where `##` is a two-digit number (00-99), these digits are used to determine the position of this item in the menu and the prefix does not appear in the menu item's name.
- A name `##`)-*** will appear as a menu separator, again with its order determined by the two-digit number `##`.

Here are the actions you can perform with the menu items in the Clippings menu:


- Choose a menu item to insert that clipping into the current script window.
- Hold down the Option key while choosing a menu item to open that clipping for editing.
- Hold down the Shift key while choosing a menu item to reveal the clipping file in the Finder.





Scripts Menu



The Scripts menu () accesses auxiliary scripts. Each menu item represents a file (or folder) in the Scripts folder.

Scripts to go in the Scripts menu should live in `~/Library/Application Support/Script Debugger 4.5/Scripts/`. Alternatively, they can live in the top-level `/Library/Application Support/Script Debugger 4.5/Scripts/`. (A third possibility is that they can live in a folder called *Scripts* in the same folder as the Script Debugger application, but this option is for historical reasons and is not recommended.)

A file will appear as a menu item. A folder will appear as a hierarchical menu, and the files inside it will be its menu items. The name of a file (or folder) is the name that will appear in the menu, except that certain names or part-names are hidden and used for determining the order of the menu, as follows:

- If a name starts with the prefix `##` , where `##` is a two-digit number (00-99), these digits are used to determine the position of this item in the menu and the prefix does not appear in the menu item's name.
- A name `##`)-*** will appear as a menu separator, again with its order determined by the two-digit number `##`.

Here are the actions you can perform with the menu items in the Scripts menu:

- Choose a menu item to run that script.
- Hold down the Option key while choosing a menu item to open that script file for editing.
- Hold down the Shift key while choosing a menu item to reveal the script file in the Finder.

Scripts to go in the Scripts menu may be AppleScript scripts (or scripts in some other OSA language, if you have any); they may also be shell scripts, applications, or Automator workflows.

A script in the Scripts menu can [drive Script Debugger itself](#). Such a script does not need to include a tell block targeting Script Debugger; Script Debugger is implicitly the tell target.

You can debug a script in the Scripts menu, if it is an AppleScript script, by opening it in Script Debugger and putting it into [debug mode](#). Now when you choose the script from the Scripts menu to run it, it will pause at a breakpoint if there is one.

If a script initiated from the Scripts menu takes a long time to execute, a progress dialog appears. This dialog contains a Stop button that you can use to abort the script if you think something has gone wrong (or if you just don't feel like waiting — you cannot do anything else in Script Debugger while a script is running from the Scripts menu).

 [Clippings Menu](#)

[Help Menu](#) 



Help Menu



Script Debugger Help. Opens this help documentation in Apple's Help Viewer application.

Script Debugger Getting Started Guide. Opens a narrative introduction to Script Debugger.

Send Us Email. In your preferred email program, creates a new email message addressed to Late Night Software.

Check For Updates. Goes online to check whether there is a more recent release of Script Debugger. A [Software Update preference](#) allows you to set this action to be performed automatically at fixed intervals.

Visit Our Website. In your browser, opens the main Late Night Software web page.

Register Your Copy of Script Debugger. In your browser, opens a Late Night Software web page where you can register to receive technical support and to be notified of news and updates.

Script Debugger Web Page. In your browser, opens the main Late Night Software web page about Script Debugger.



[Scripts Menu](#)



Preferences



When you choose the Preferences menu item from the application (Script Debugger) menu, you summon the Preferences window. It has eight preference panes. These pages describe the options on each pane.

To **navigate to any preference pane**, click Show All in the toolbar. Now you can click a pane's icon to navigate to it.

The [toolbar](#) in the Preferences window is customizable. Do not accidentally close the toolbar, since without it you cannot access the Show All button and you will have no way to navigate to the different preference panes.

The **Factory Defaults** button on each pane sets the options *in that pane* to the Late Night Software default values.

All changes to preference options take effect immediately — except for the [Fonts & Colors preferences](#), which have to be set with the Apply button (because in this case, you're talking to AppleScript, not to Script Debugger).

Further Details:

- [Preferences: General](#)
- [Preferences: Key Bindings](#)
- [Preferences: Editor](#)
- [Preferences: Text Substitutions](#)
- [Preferences: Fonts & Colors](#)
- [Preferences: Debugger](#)
- [Preferences: Dictionary](#)
- [Preferences: Software Update](#)



Preferences: General



The General [preferences pane](#) collects a number of options having mostly to do with Script Debugger's startup behavior and how Script Debugger opens and saves [script files](#).

On Startup:

Remember open scripts. If checked, then when Script Debugger quits, all open scripts are remembered and will be reopened automatically the next time Script Debugger starts up.

Remember open dictionaries. If checked, then when Script Debugger quits, all open dictionary windows are remembered and will be reopened automatically the next time Script Debugger starts up.

Create new script if nothing else is open. If checked, then when Script Debugger starts up, if no other window opens, a new script window will be created.

On Reopen:

Create new script if nothing else is open. If checked, then when Script Debugger gets a Reopen event, if no window is open, a new script window will be created. A Reopen event is sent, for example, when you click on Script Debugger's Dock icon (but *not* when you press Command-Tab to switch to Script Debugger).

Saving:

Script Debugger is always creator vs. **Keep original creator** vs. **No creator.** The question here is, when you open a saved script file in the Finder, [what application should open the file](#).

Keep backup files. If checked, then just before saving an already existing file, a copy of the currently saved version of the file will be saved as *Filename~* (the same name as the file, plus a tilde character).

Opening:

Remember Result drawer state. If checked, then when Script Debugger opens a script file, it opens the script's [result drawer](#) if the file was previously saved by Script Debugger with the drawer open. Otherwise, saved script files will be opened with their result drawer hidden.

Warn when applications may be launched. If checked, then when Script Debugger begins opening a script file, it puts up a "Launch Applications?" [dialog](#) if continuing to open the file might cause AppleScript to [launch](#) an application targeted in the script.

Mac OS Settings:

Respond to applescript:// URLs in web pages. The [applescript protocol](#) permits a hyperlink (in a web browser, a PDF document, and so forth) to contain AppleScript code, to be displayed by a script editor application when the link is clicked. (The script editor application does not automatically *run* the code, as that would be a security violation.) By default, the protocol sends its messages to Apple's Script Editor, and Apple provides no interface for changing this target. This checkbox is provided so that you can switch the routing of the protocol to Script Debugger.

Default editor for OSA scripts, applets, and droplets. Mac OS X may [ignore](#) a file's creator code and determine from the filename extension what application opens the file. This checkbox lets you associate the relevant filename extensions (*.scpt* and so on) with Script Debugger. (It also causes Script Debugger to be the editor that responds to the Edit button in an [applet's](#) runtime error dialog.)

In theory, you could accomplish the same thing by choosing Script Debugger in the Default Script Editor popup menu of Apple's own AppleScript Utility. However, the AppleScript Utility's behavior may be buggy — for example, it may incorrectly associate all text files with your chosen default script editor — so its use is not recommended.

[Preferences: Key Bindings](#) 



Preferences: Key Bindings



Script Debugger permits you to customize (change) the keyboard shortcut for any menu item, including [Clippings](#) and [Scripts](#). The Key Bindings [preferences pane](#) is where you do that.

The table lists all of Script Debugger's menus and menu items, in hierarchical outline format. By clicking the disclosure triangles (or by double-clicking a line), drill down to the menu item whose keyboard shortcut you'd like to alter. Click the Set button, or double-click the menu item listing. The Keystroke dialog opens.

With the Keystroke dialog showing, type a keyboard shortcut. It must involve at least the Command key or the Control key, or be a Function key (F1, F2, etc., plus Home, Page Up, and so on). It may additionally involve any combination of modifier keys (Shift, Control, Option, Command). Script Debugger warns you if you type a keyboard shortcut that is already in use by another menu item.

To remove an existing keyboard shortcut from a menu item, so that that item has no keyboard shortcut, select it and click Clear.



Preferences: Editor



The Editor [preferences pane](#) collects a number of options having mostly to do with Script Debugger's behavior as you type, as well as certain appearance settings in [script windows](#) and other windows.

Editing Options:

Auto indent. If checked, then when you create a new line in a script window by typing Return, its indentation will match the indentation of the preceding line. Otherwise, the new line will start at the left edge of the window. (AppleScript will indent properly in any case when the script is compiled.)

Auto-pair delimiters ([{ " " }]). If checked, turns on Script Debugger's [auto-pairing](#) feature.

Auto-close AppleScript blocks (end tell, etc.). If checked, turns on Script Debugger's [auto-closing](#) feature.

Paste Object References as nested Tell blocks. If checked, then when pasting an object reference (such as you might obtain by copying from an [explorer view](#)), what's pasted is a nest of tells. If unchecked, what's pasted is a single line of ofs. Hold down the Option key as you choose Edit > Paste to reverse the behavior from your preference here.

Share Find string with other applications. Cocoa maintains a "Find panel pasteboard" where all applications can share their Search For text. This means that if you search for text in one application (say, Safari) and then switch to another application (say, TextEdit) and bring up the Find dialog, the very same search text is present. This behavior can be beneficial or annoying, so this checkbox lets you turn this feature on or off.

Synchronize split-view appearance. If checked, then changing a [view](#) setting in a [split view pane](#) changes the same setting for the other panes of the same script. Hold down the Option key as you change a view setting to reverse the behavior from your preference here.

Balance includes enclosing ([{ }]) delimiters. If checked, then the [balance command](#) selects everything *including* the delimiters surrounding the starting selection; otherwise, it selects everything *enclosed by* the delimiters surrounding the starting selection.

Auto-hilite opening ([{ when typing closing }]). If checked, then when you type a [right delimiter](#), the corresponding left delimiter is momentarily highlighted (and if there isn't one, Script Debugger beeps).

- **Hilite delay.** Sets the length of time during which the momentary highlighting is present.
- **Scroll if necessary.** If checked, Script Debugger will scroll backwards if necessary to reveal the highlighted left delimiter.

New Line Character:

Sets the [line-end character](#) that is typed in a script when you press the Return key.

Tab Width:

Sets the number of spaces to which a tab character should be equivalent. This is how far a nested block is indented in a compiled script, and how many spaces a tab character is converted to when you choose [Edit > Entab](#).

Table Of Contents Options:

Sort table of contents menu alphabetically. If checked, the [table of contents menu](#) is sorted alphabetically. Otherwise, its order is the order in which things appear in the script. Hold down the Shift key while summoning the table of contents menu to see it sorted in the order opposite to your preference here.

Show navigation bar location when scrolling. If checked, then as you scroll a script window, a tooltip appears showing the line number of the line currently appearing at the top of the window, along with other [navigation bar](#) information about that line.

Edit > Comment Inserts:

Sets the string prefixed to the start of each selected line by the [Edit > Comment](#) command.

Background color:

Sets the background color for [script windows](#), [viewers](#), and the [Apple Event Log window](#).

Cursor color:

Sets the color of the thin insertion point cursor for [script windows](#), [viewers](#), and the [Apple Event Log window](#). (To set the color of a text selection comprising one or more characters, use System Preferences > Appearance > Highlight Color.)

AEPrint color:

Sets the color of [AEPrint](#) text in [viewers](#) and the [Apple Event Log window](#) (as well as a few types of [Best](#) text whose color is not set by the [Fonts & Colors Preferences](#)).

Highlight line containing insertion point. If checked, the entire line in a script window containing the insertion point is banded in yellow for greater visibility.

Highlight block when mouse hovers in gutter. Turns on or off the [block highlighting](#) feature.

Show compiled state in gutter. If checked, then a script that needs compilation has a [stripy pattern](#) in its gutter.

Font sizes:

Result/Logging. Sets the font size for [viewers](#), [explorers](#), the [call stack](#), [expressions](#), [logging entries](#), and the browser at the top of the [dictionary](#) window. (The font size for script windows is a [Fonts & Colors preference](#).)

Inspectors. Sets the font size for [inspectors](#).



[Preferences: Key Bindings](#)

[Preferences: Text Substitutions](#)





Preferences: Text Substitutions



The Text Substitutions [preference pane](#) is where you manage [text substitutions](#).

Automatic Substitution:

Enabled. If checked, the substitution feature is turned on.

If substitution is turned on, then when you type an enabled “Replace” column entry followed by a non-word character (such as a space or a Return), the corresponding “With” column entry will be substituted for it.



[Preferences: Editor](#)

[Preferences: Fonts & Colors](#)





Preferences: Fonts & Colors



The Fonts & Colors [preferences pane](#) has to do with AppleScript formatting, that is, the pretty-printing of [compiled scripts](#). This facility is provided by AppleScript (as part of the [decompilation](#) process), not by Script Debugger. This preference pane thus accesses AppleScript's preferences.

You can select multiple lines of the table and change their font or color all at once.

Changes are not sent to AppleScript *until you click the Apply button*. When you do, any compiled scripts that are open now, as well as any compiled script files that you open or create in the future, will take on the formatting you have specified. To cancel (changing your mind without applying your changes), switch to another pane, or close the window.



Preferences: Debugger



The Debugger [preferences pane](#) collects a number of options having to do with Script Debugger's behavior when [running](#) and [debugging](#) scripts.

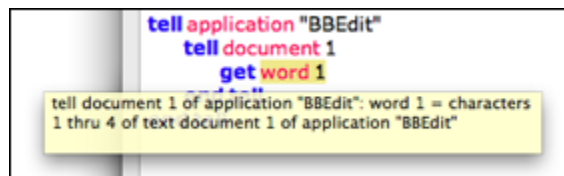
Debugging Options:

Pause script when external debugging begins. If checked, then when [external debugging](#) starts (that is, a script saved in [debug mode](#) starts to run in some other context), Script Debugger will pause before the first executable line of the script.

Restore active application when stepping/continuing. If checked, then when you resume execution after a pause, Script Debugger switches to the application that was active before the pause.

(This preference is ignored if the script is being controlled from Script Debugger's Dock menu.)

Show expression tooltips. If checked, then when you hover the mouse over text in a script window, the AppleScript expression under the mouse will be evaluated and its value shown in a tooltip. In some cases, it will help to select the desired expression first and then hover the mouse over it. For your safety, Script Debugger prevents evaluation of an expression if evaluating it takes a long time or might have major side effects like deleting or altering an object (such an expression will generate no tooltip).



- **Tooltip delay.** Sets the delay between the time when the mouse hovers over an AppleScript expression and the time when the expression is evaluated.
- **Show only while debugging.** If checked, then these tooltips appear only when the script is [paused](#) in [debug mode](#).
- **Include tell context.** If checked, then if there is a [tell context](#), it is included at the start of the tooltip (as in the above illustration — if this option were not checked, everything up to the colon would be absent).

Bring Script Debugger to foreground when scripts pause. If checked, then when [pausing](#) in debug mode, Script Debugger will come to the front.

Bring Script Debugger to foreground when scripts end. If checked, then any time a script finishes executing, Script Debugger comes to the front.

(This preference is not limited to debugging. It is ignored if the script is being controlled from Script Debugger's Dock menu.)

Show result when scripts pause or end. The question here is what should happen with regard to the [display of the result](#) when a result is produced.

(This preference is not limited to debugging.)

Script Error Actions:

Bring Script Debugger to foreground. If checked, then Script Debugger comes to the front when it puts up a [runtime error dialog](#). Otherwise, Script Debugger bounces the notification icon in the Dock.

Beep. If checked, then Script Debugger beeps when it puts up a [runtime error dialog](#).

 [Preferences: Fonts & Colors](#)

[Preferences: Dictionary](#) 



Preferences: Dictionary



The Dictionary [preferences pane](#) collects a number of options having to do with the appearance and behavior of [dictionary](#) windows and [explorer](#) views.

Opening:

Governs what should happen when you [open a dictionary window](#). The issue here is that a dictionary window has two panes: the [dictionary](#) and the [explorer](#). Which one should appear when the window opens? Your choices are:

- **Show Dictionary.** The Dictionary pane appears.
- **Show Explorer.** The Explorer pane appears.
- **Remember Explorer/Dictionary state.** The pane that appears is the pane that was showing when the dictionary window for this application was closed previously.

Explorer Options:

Scan for elements if count fails. Some badly behaved applications do not implement count properly, so Script Debugger can't learn how many elements there are, and can't populate the hierarchical display of those elements. (Eudora is a notorious case in point.) In such cases, if this option is checked, Script Debugger will ask for elements by index until a runtime error is encountered in order to discover how many there are.

Show list and record items for expanded elements. If checked, lists and records will be expandable in the explorer in certain rare cases where they normally would not be (most of the time, they will be anyway).

Show contents of list and records. Controls the "value" shown for lists in the explorer. If unchecked, the value tells the size of the list. If checked, the value is the literal list, i.e. curly braces containing items separated by commas.

Show description tooltips. If checked, tooltips appear when the mouse is hovered over items in the left column of the explorer. These tooltips are the comment from the dictionary describing the entity in question.

Show value tooltips. If checked, tooltips appear when the mouse is hovered over items in the right column of the explorer. These tooltips are the comment from the dictionary describing that class or enumeration. Not every item in the explorer has such a comment, in which case there is no tooltip.

Outline Options:

Show hidden items. If checked, hidden items are displayed in the dictionary. An example is the `computer` command in *StandardAdditions*. Hidden items are generally hidden by the application's developer for a good reason, and showing them is not recommended.

Dictionary Caching:

Cache generated dictionaries. If checked, Script Debugger maintains cached copies of application [dictionaries](#), for faster display. If unchecked, Script Debugger may be considerably slower when it needs to open or search a dictionary. Most users should not need to uncheck it. Concerning why you *might* want to uncheck it, or why you might want to click **Clear Cache**, read [here](#).



[Preferences: Debugger](#)

[Preferences: Software Update](#)





Preferences: Software Update



The Software Update [preference pane](#) lets you specify whether Script Debugger should periodically go online to check for a more recent version of itself, and if so, how often. You can also click the Check Now button to check manually (the same functionality is available from the [Help menu](#)).

Windows

Script Debugger's window types may be categorized as follows:

- [Script windows](#)
- [Dictionary windows](#)
- [Viewer windows](#)
- The [Apple Event Log window](#)
- [Inspectors](#)
- The [Preferences window](#)

Details about [inspectors](#) are provided here.

Further Details:

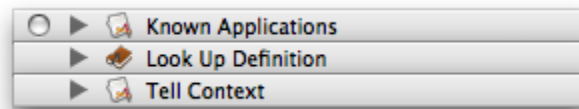
[Inspectors](#)

Inspectors

Inspector windows are summoned from the Window > Inspectors menu. They have several features in common:

- They float (they are in front of all other windows), and they are utility windows (they vanish when Script Debugger is in the background).
- They have two states, expanded or contracted. Click an inspector's title bar to toggle its state, which is also reflected by a disclosure triangle. When contracted, an inspector consists of *just* its title bar. This allows the inspector to be positioned unobtrusively.
- They can be docked together, and separated again. To dock one inspector to another, drag it to a position just above or below the other inspector. It will snap into place, and the inspectors will share a single close button, showing that they are docked together.

Docked inspectors can be expanded or contracted separately without losing the docked arrangement. Otherwise, they act like a single window. If you drag the topmost inspector's title bar, the whole docked set is repositioned. If you close (dismiss) the topmost inspector, the whole docked set is closed. If you summon an inspector that has been dismissed as part of a docked set, the whole docked set appears. To separate docked inspectors, drag the lower inspector's title bar.



The above illustration shows three inspectors, contracted and docked together.

To **summon an inspector**:

- Choose Window > Inspectors and the desired inspector.

To **close (dismiss) an inspector**:

- Click the close button in its title bar. Alternatively, you can hide *all* inspectors by choosing Window > Inspectors > Hide Inspectors. In this case, you can summon all inspectors that were hidden, by choosing Window > Inspectors > Show Inspectors.

To **restore inspectors to the default arrangement**:

- Choose Window > Inspectors > Reset Inspector Locations.

Further Details:

[Known Applications Inspector](#)
[Look Up Definition Inspector](#)
[Tell Context Inspector](#)
[Clippings Inspector](#)
[Scripts Inspector](#)
[Windows Inspector](#)



Known Applications Inspector



The Known Applications inspector lists applications that Script Debugger has “met” in various ways. This list appears in several places in the interface — for example, in the File > [Open Dictionary](#) menu and the Edit > [Paste Tell](#) menu. From the Known Applications inspector, you can manipulate the list, as well as perform the two main actions that the list is good for.

- To [open an application’s dictionary](#), select it and click the Dictionary button, or double-click the application’s name.
- To [insert a tell block](#) targeting an application, select it and click the Paste Tell button. Hold down the Option key to create a new script window at the same time.

Alternatively, drag the name of an application from the Known Applications inspector into a script window.

- To **learn an application’s location**, hover the mouse over it to see the tooltip.
- To **show an application in the Finder**, select it and choose Reveal Application from the tool popup in the upper right corner (or the contextual menu).
- To **remove an application from the list**, select it and choose Forget Application from the tool popup (or use the contextual menu). (You can also choose Forget All Applications to empty the list.)
- To **add an application manually to the list**, choose Add Application from the tool popup (or the contextual menu).
- To **toggle the visibility of icons in the list**, choose Hide / Show Icons from the tool popup (or the contextual menu).

The list also provides access to AppleScript *commands* that you can use in your script when targeting each application. To see them, click the disclosure triangle next to the application’s name. Then:

- To **look up a command’s definition in the dictionary**, select it and click the Dictionary button, or double-click the command.
- To **insert a command template** into your script (including a tell block, if necessary), select the command and click the Paste Tell button (or drag the command from the Known Applications inspector into a script window). Hold down the Option key to create a new script window at the same time.

You may be curious about how the Known Applications list is automatically populated. The answer is a little complicated, but here’s the short version. An application is added to the list automatically:

- When Script Debugger is first installed (it looks on your computer for certain “well known” scriptable applications, and adds them to the list if they are present).
- When an application’s dictionary is explicitly opened.

- When an object specifier is explored (so, for example, when you target an application in a script and it returns an object as the script's result).
- When a tell block targeting that application is detected in your code.

Running a script targeting an application might not add that application automatically to the list. The reason is that AppleScript and Script Debugger are two different entities, so AppleScript can run a script without Script Debugger seeing and analyzing the contents of that script.

[Look Up Definition Inspector](#) 



Look Up Definition Inspector



The Look Up Definition inspector is discussed [here](#).



[Known Applications Inspector](#)

[Tell Context Inspector](#)





Tell Context Inspector



The Tell Context inspector is a live [explorer view](#), showing the current elements and properties of the application or object targeted at the point where you are working in your script (the current [tell context](#)).

The Tell Context inspector drills down along with your script into successively deeper levels of tell context. For example, consider this script:

```
tell application "BBEdit"
  tell document 1
    get word 1
  end tell
end tell
```

If you select "document" in that script (in the second line), the Tell Context inspector shows the elements and properties of BBEEdit (the application object). If you select "word" (in the third line), the Tell Context inspector shows the elements and properties of document 1 of application "BBEdit".

If you open the Tell Context inspector and it is empty or disabled or otherwise looks wrong, try switching explicitly to the window that you want inspected, and if necessary, compile it. This should wake up the Tell Context inspector and get it in synch with your activities in the script window.

The Tell Context inspector has the same basic functionalities as a dictionary [explorer](#), but some of them are accessed a little differently because this is a floating palette (menus in the menu bar don't apply to it, and keystrokes don't target it):

- To **insert a reference** to an attribute into the frontmost script, click the Paste button. You can also drag from the inspector into your script.
- To **open the dictionary** for the currently targeted application (its name appears at the top of the inspector), click the Dictionary button.
- To **reload** the data for the currently selected line, click the Reload button. Hold down the Option key to reload *all* the data in the inspector.
- To **edit the value** of a property, choose Edit Value from the contextual menu.
- To **generate a separate [viewer window](#)** for a value, choose Open Viewer from the contextual menu.

Other [explorer functionality](#) works normally.



Clippings Inspector



The Clippings inspector is a convenient way to [work with clippings](#).

- To **insert a clipping**, double-click it, or select it and click the Paste button.
- To **edit a clipping**, hold down the Option key and double-click it, or select it and click the Edit button.
- To **show a clipping file**, hold down the Shift key and double-click it, or select it and choose Reveal In Finder from the tools popup (or the contextual menu).



[Tell Context Inspector](#)

[Scripts Inspector](#)





Scripts Inspector



The Scripts inspector is a convenient way to work with scripts from the [Scripts menu](#).

- To **run a script**, double-click it, or select it and click the Run button.
- To **edit a script**, hold down the Option key and double-click it, or select it and click the Edit button.
- To **show a script file**, hold down the Shift key and double-click it, or select it and choose Reveal In Finder from the tools popup (or the contextual menu).



[Clippings Inspector](#)

[Windows Inspector](#)





Windows Inspector



The Windows inspector gives access to all open windows. In some ways it's like the [Window menu](#) — it uses the same icons to reflect a window's status, for example — and at the same time it permits a number of fundamental operations that can also be performed from elsewhere.

Thus, you can Close, Save, Print, Run, and Stop a window. Additionally:

- To **close all windows**, hold down the Option key as you click Close.
- To **trace instead of running**, hold down the Option key as you click Run.
- To **reveal a script window's file in the Finder**, select it and choose Reveal In Finder from the tools popup (or the contextual menu).



[Scripts Inspector](#)

Glossary

This section consists of a series of short pages explaining some words and concepts used in this help document.

Further Details:

[Glossary: Bundle](#)
[Glossary: Bytecode](#)
[Glossary: Compiled Script File](#)
[Glossary: Dictionary](#)
[Glossary: Fork](#)
[Glossary: Object Model](#)
[Glossary: Scripting Addition](#)
[Glossary: Sdef](#)
[Glossary: Tell Context](#)



Glossary: Bundle



A **bundle** (or **package**) is a file system entity in Mac OS X whose key characteristic is that although it is a folder, it is portrayed in the Finder as a file. Bundles are useful because they can contain files and folders inside them which the user doesn't see or even know about. In fact, the user is generally unconscious of the fact that a bundle is a bundle. Just to give a simple example, a Mac OS X-native application is a bundle.

Opening a bundle in the Finder by double-clicking it is like opening an application or document - not like opening a folder. If you want to open a bundle as a folder, control-click the bundle in the Finder, and choose Show Package Contents from the contextual menu.

Inside a bundled compiled script or bundled application, when you Show Package Contents, is a Contents folder. Inside that is a Resources folder. That is where you can keep files that need to travel with the bundle.

Do *not* touch any of the other files and folders inside the Resources folder, or disturb anything else in the bundle. Doing so can destroy the viability of the bundle.
Do *not* modify the bundle contents while editing the script, as this may prevent saving of the script or have other unwanted consequences.

A bundled compiled script or a bundled application can refer to a file inside its Resources folder using the `path to resource` scripting addition command.

Every time you save a bundled compiled script or bundled application, the contents of the Resources folder are duplicated and saved anew, as a safety measure. This will alter the modified date of your resource files.





Glossary: Bytecode



AppleScript code is [compiled](#) into **bytecode**, meaning that, roughly speaking, the nouns and verbs of the original text are translated into a sort of compressed, coded equivalent, called *tokens*. These tokens are meaningful to the AppleScript runtime engine (and illegible to everyone else). The runtime engine interprets the bytecode, parsing whatever tokens it meets along its path of execution, accumulating them into chunks, and translating these chunks further, as necessary, in order to execute them.

The implication for you, the AppleScript programmer, is that in its compiled form, a script is illegible. So why are you able to read a [compiled script file](#)? It's because AppleScript *decompiles* the bytecode, translating the tokens back into their English-like form. If the script targets an application, this decompilation requires the application's [dictionary](#). This is one reason why things can go wrong when you attempt to open a compiled script. (Script Debugger may be able to help in such a situation by letting you [open the script as text](#).)

Actually, a compiled script file contains not only bytecode but also some further information (such as variable names) needed to decompile the tokens. In a [run-only script](#), this further information is not present, which is why the script cannot be decompiled (and therefore cannot be read by a human being).



Glossary: Compiled Script File



The fundamental AppleScript file format is the [compiled](#) script file. It consists of [bytecode](#), not the original text. It also can maintain other information, such as the persistent values of top-level entities (mostly properties, globals, and script objects) and certain context information. A compiled script file can be executed directly, with very little delay (because there is no need to compile). Many environments that can run scripts expect a compiled script file.

When you run a compiled script in Script Debugger and then save (without editing further), [values of top-level entities](#) are saved as well.

The life of the AppleScript programmer is made more complicated by the fact that compiled script files now come in a [variety of formats](#). Not all of these are compatible with every system or every script-editing or script-running environment.

Also, a compiled script can have [difficulty opening](#) if a [required application or scripting addition](#) is missing. If the script was saved with Script Debugger, you may still be able to [open the script as text](#).



Glossary: Dictionary



AppleScript's real power and purpose lies in communicating with scriptable applications, which provide powers that AppleScript lacks. In order that you, the AppleScript programmer, may harness its powers, a scriptable application extends the vocabulary of the AppleScript language. This extended vocabulary is called a scriptable application's *terminology*. A *dictionary* is the means by which a scriptable application or scripting addition lets you (and AppleScript) know how it extends AppleScript's vocabulary.

The dictionary translates between two forms of terminology — the *English-like terms*, which you use in your script, and the *raw Apple event codes*, which AppleScript uses to construct Apple events when communicating with a scriptable application.

When you write a script using English-like terms, the dictionary is used to translate them into raw Apple events to be sent to scriptable applications.

In a [compiled script file](#), the raw Apple events are encoded directly into the [bytecode](#). In order to open the compiled script file and decompile it, the dictionary is used to translate the raw Apple events back into English-like terms. This is why AppleScript may have [trouble opening a script](#) in the absence of a [required dictionary](#).

Under certain circumstance, a compiled script may open but display some of its raw Apple events (as four-letter codes surrounded by chevrons) instead of the English-like terminology. And, as a power user feature, Script Debugger lets you *deliberately* [display raw Apple events](#) instead of English-like terminology.

```
tell application "Finder"
  make new «class pRSN» with properties
end tell
```

At any given moment in a script there are several sets of terminology visible at once — variable names used in the script, terms from the dictionary of the [targeted application](#), terms from [scripting additions](#), terms from AppleScript's own dictionary. If the script chooses its terms unwisely, or if the visible dictionaries use the same terminology in different ways, *terminology clash* can occur. Terminology clash can result in a script that won't compile or run, or it might result in a script that compiles and runs, but behaves mysteriously.

Script Debugger helps you track down terminology clash by letting you view raw Apple event codes in your [script](#), in the [Apple Event Log window](#), and in [dictionaries](#), and by letting you [search for terminology](#) in all visible dictionaries at once.

If you use an application with an 'aete' dictionary that allows the dictionary to be extended through plug-ins (such as QuarkXPress or InDesign), read the discussion of Script Debugger's [dictionary caching](#) mechanism.



Glossary: Fork



In the earliest days of the Macintosh file system, a structure was devised whereby a file could have two pieces, the data fork and the resource fork. The data fork was a single thing, and was just for data (like the text of a TeachText / SimpleText file). The resource fork was for secondary information, and could contain many resources, accessible by category and name or number, as in a kind of miniature database (so, for example, style information in a TeachText / SimpleText file).

When Mac OS X was introduced, Apple undertook a concerted effort to deprecate the resource fork, because it wasn't a standard Unix file system thing (in fact, a Macintosh file moved to another file system, such as Windows or true Unix, will usually lose its resource fork). As part of this effort, a new format for [compiled script files](#) was devised, where the [bytecode](#) was kept in the data fork instead of the resource fork.

Ironically, Apple has recently realized that resource forks are a good thing (because they provide a place to put file metadata) and has reversed course — they've modified the file system so that a file can now (starting with Tiger, Mac OS X 10.4) have any number of extra forks.



Glossary: Object Model



A scriptable application can define classes, roughly comparable to the “things” that constitute its world. (For example, iTunes defines a `playlist` class and a `track` class, and the Finder defines a `folder` class and a `file` class.) Classes can have properties and elements, whose value type can be a class. Thus, in theory, for a given application, there is a hierarchical relationship of ownership—of containment—amongst its classes. (For example, in iTunes, a playlist “has” tracks, and in the Finder, a folder “has” files.)

In theory, this relationship can be expressed as a “tree”, a hierarchical structure starting with the application itself, and containing every object in the application. Indeed, this tree is vital to your use of AppleScript to communicate with a scriptable application, because it is why you are able to refer to an object in the first place. (For example, you can speak to the Finder of `file 1 of folder "Mannie"` because the Finder “has” folders and a folder “has” files.) This tree of the objects that you can refer to is the application’s **object model**.

Script Debugger exposes an application’s object model in two places:

- In a Dictionary window, the [diagram drawer](#) shows you the application’s containment hierarchy as a tree.
- In a Dictionary window and in other places, an application’s actual objects are displayed hierarchically through an [explorer view](#).

I say “in theory” because in reality things are not so simple. The containment hierarchy is describing what’s possible, not what’s real, and therefore infinite recursions and circularities can result when you try to express it as a simple tree. For example, in the Finder, in real life there could never be an infinite depth of folder containment. In the [containment hierarchy diagram](#), however, you can keep opening “folder” entries to get “folder” entries inside them, and so on, forever and ever (or until you get bored), just because as a theoretical matter, it’s always true that a folder can contain a folder. If this confuses you, and you’d rather see the object model as it exists in reality, use the [explorer](#).



Glossary: Scripting Addition



A **scripting addition** is a compiled-code resource that implements additional AppleScript language terminology and commands. AppleScript loads any scripting additions that it finds in the *ScriptingAdditions* folder (in */System/Library*, in */Library*, or in your user *Library*), and the terms and commands implemented in them become effectively part of the language.

Because of the nature of this mechanism, you don't have to be in any application's [tell context](#) in order to use a term defined by a scripting addition. Instead, a scripting addition's terms are simply "part of the language". For example, to learn today's date, you just say `current date`, anywhere. You do not — indeed, you cannot — explicitly target a scripting addition.

Nevertheless, everything in AppleScript works through dictionaries, so every scripting addition has a [dictionary](#). Script Debugger collects all the dictionaries of all loaded scripting additions into a [single dictionary display](#), called "Scripting Additions".

Because scripting addition terminology is always available, no matter what application you are talking to (or even if you are talking to no application), Script Debugger makes scripting addition terms automatically part of the [tell context](#).

Scripting additions are a notorious source of [terminology clash](#).

Running a script in the absence of a scripting addition on which the script depends will probably result in a mysterious [runtime error](#). Opening such a script will result in the appearance of raw Apple event codes in the script (Script Debugger tries to [help track down](#) the source of the problem, but it isn't easy because the name of the missing scripting addition doesn't appear in the code, since scripting additions are not targeted).



Glossary: Sdef



An “sdef” (pronounced “ess-deaf”, and standing for “scripting definition”) is a new [dictionary](#) format, starting in Tiger (Mac OS X 10.4). Script Debugger takes advantage of this new format. Relatively few applications actually use this new format, though, so Script Debugger translates the old format (called an ‘aete’) into the new format before presenting the dictionary to you. As the new format comes into more common use, it will be possible for application authors to provide more accurate, informative dictionaries.

If you use an application with an ‘aete’ dictionary that allows the dictionary to be extended through plug-ins (such as QuarkXPress or InDesign), read the discussion of Script Debugger’s [dictionary caching](#) mechanism.



Glossary: Tell Context



The purpose of AppleScript is to communicate with scriptable applications. The linguistic construction commonly used to perform such communication is the *tell block*. For example:

```
tell application "TextEdit"
    activate
    set word 1 of document 1 to "Hello"
end tell
```

In that code, the indented lines are within a tell block targeting TextEdit. Thus, TextEdit is their *tell context*.

Nested tell blocks can narrow the tell context still further. For example:

```
tell application "TextEdit"
    activate
    tell document 1
        set word 1 to "Hello"
    end
end
```

In that code, the set line's tell context is document 1 of TextEdit.

Script Debugger watches as you work, and knows what the tell context of the current selection or insertion point is. This knowledge is fundamental to certain Script Debugger features. For example:

- The [Tell Context inspector](#) explores the elements and properties of the targeted application relative to the current tell context.
- The [Look Up Definition inspector](#) can search in the current tell context. So, if you select "word" in the above code and choose Search > Look Up Definition, and if the Look Up Definition inspector is set to Search in Tell Target, Script Debugger will search for the term word in TextEdit's dictionary.
- The [File > Open XXX Dictionary](#) menu item knows what dictionary to open based on the tell context.

The tell context itself may depend upon the value of a variable at a given moment. Script Debugger handles this situation correctly, though the results may surprise you. For example:

```
on doTell(x)
    tell application x
        get window 1
    end tell
```

```
end doTell  
doTell("Finder")
```

Suppose this script is paused (in debug mode) at the line `get window 1`. What should the tell context be if we click in that line? Well, there are two [stack frames](#): the `doTell` handler call, and the top level of the script (the implicit run handler). In the stack frame of the `doTell` handler call, `x` has a value and there is a tell context (the Finder). But in the stack frame of the top level of the script, `x` has no value and there is no tell context (and the Tell Context inspector will report “no selected tell block”). So the tell context depends upon the selected stack frame as well as the current selection.

 [Glossary: Sdef](#)



Frequently Asked Questions



This section provides answers to some questions that were too technical or too miscellaneous to incorporate into the main discussion.

Note: By its very nature, information in this section may be volatile. For the latest facts, check at [Late Night Software's web site](#).

Further Details:

[What's Installed Where?](#)

[Is Script Debugger's AppleScript the Same as Script Editor's?](#)

[What's The Big Deal With Line Endings?](#)

[Why Do Applications Open Spontaneously?](#)

[Hey, Script Debugger Changed My Formatting!](#)

[Why Doesn't My Script Debug Properly?](#)

[How Do I Script Script Debugger?](#)



[Glossary](#)



What's Installed Where?



Script Debugger is a good Mac OS X citizen, and does not heedlessly install files in places where it's not supposed to. Still, it does install files in a variety of locations, and you might wish to know where these are and what the files do. So here's a list.

~/Library/Components

Script Debugger installs *Script Debugger.component* here. It implements [debugging](#), by means of an [OSA language](#) which you can see listed in the Language popup menu in any Script Debugger script window (or any Script Editor script window, for that matter).

~/Library/Caches/Script Debugger 4.5

Script Debugger maintains [caches](#) of application dictionaries here. These caches mean that Script Debugger can open and access dictionaries much more quickly.

~/Library/Application Support/Script Debugger 4.5

The *Clippings* folder is where Script Debugger gets the items to populate the [Clippings menu](#) and the [Clippings inspector](#). The *Scripts* folder is where Script Debugger gets the items to populate the [Scripts menu](#) and the [Scripts inspector](#). The *Script Libraries* folder is where Script Debugger will automatically look for a [library](#) to attach to a script. You can modify the contents of these folders.

Script Debugger automatically populates these folders to start with. If you need Script Debugger to perform this automatic population again, move the *~/Library/Application Support/Script Debugger 4.5* folder aside (for example, drag it to the desktop), then quit Script Debugger and start it up again. Now you can move your own materials back into place if desired.

~/Library/Preferences

Script Debugger keeps one preference file here, *com.latenightsw.ScriptDebugger.plist*. This contains a lot of information, including state information for various windows, so be aware that if you throw out this file, such information will be lost.

Also, Script Debugger maintains the *Script Debugger Preferences* folder, which contains the [default document](#) that is opened when you create a new script window.





Is Script Debugger's AppleScript the Same as Script Editor's?



From a purely linguistic point of view, Script Debugger naturally tries to be as compliant as possible with the standard behavior of the AppleScript language. After all, it wouldn't be good if you could write a script in Script Debugger which would not run properly in other environments. Sometimes, however, "standard behavior" means "buggy" or "inconvenient". Apple's Script Editor, for example, occasionally does things incorrectly, or badly, and Script Debugger, in doing better, must by definition do differently.

This page summarizes some of the AppleScript differences between Script Debugger and Script Editor.

Stop Log

The `start log` and `stop log` commands are broken in Script Editor, and have been for years. They [work](#) in Script Debugger.

Libraries

Script Debugger's [libraries](#) feature goes a long way towards solving the problem of modularizing scripts in a clean and simple way, but it isn't compatible with other script-editing environments. That is why you have to *flatten* a script that uses libraries if you intend to edit it outside Script Debugger.

Alternative OSA Languages

A script editor is supposed to be able to open a compiled script file saved in *any* [OSA language](#), not just AppleScript. Script Debugger can do this. Apple's Script Editor used to be able to do it too, but in recent versions, it no longer can. Thus, if you save a script in [debug mode](#), which uses the AppleScript Debugger X OSA language, or if you save a script in the JavaScript OSA language, Script Editor won't be able to open it.

Persistence

When a script is saved with Script Debugger, the current values of top-level properties and globals are [saved](#) along with it, and are still there when the script is opened again later. Script Editor strips these values when it opens a script.

This point is worth a little further explanation. When you open a compiled script with Script Debugger, the values of top-level entities persist from the last time the script was executed. They are not reinitialized to their base values until the next time the script is compiled. (Merely running a script, without changing it, does not compile it.)

So, for example, a script's top-level property `greeting` might be *defined* as "howdy", but it might not actually *be* "howdy" when you start running the script (unless you compile the script first), because `greeting`'s value might have been changed to something else (typically by the script itself as it ran last time), and the changed value will persist. And Script Debugger also *shows* you what the value *really* is, in the [variables pane](#) of the script window's result drawer.

But if you so much as open the script with Script Editor, the persistent value is stripped out, so now when the script is run, `greeting` is reinitialized to "howdy".

Note: If AppleScript refuses to save your compiled script, generating a "Stack Overflow" error, the cause might be an AppleScript bug connected with persistent top-level values. Force the script to [recompile](#) to delete the persistent top-level values, and try saving again.



[What's Installed Where?](#)

[What's The Big Deal With Line Endings?](#)





What's The Big Deal With Line Endings?



AppleScript makes line endings a complicated issue. Script Debugger does a lot to make them simple, or at least transparent. But sooner or later you may run into problems with line endings, so here's an explanation.

Imagine that two mighty forces are at work, tussling with line endings:

- **The AppleScript compiler.** AppleScript is a [compiled](#) language, and this compilation involves a transformation performed directly on your code. Your code starts life as ordinary text, but when it is compiled, your file becomes a [compiled script file](#), and you are shown the decompiled [bytecode](#). The AppleScript compiler goes back to the early 1990s, a time when Macintosh line endings were all CR (ASCII 13). So, after compilation, every complete line of code in your compiled script file ends with a CR.
- **Unix.** Mac OS X is Unix, and the standard Unix line-end character is LF (ASCII 10). For example, a script written in a Unix scripting language such as Perl or Ruby needs to have LF line endings or it may not run properly. Many Mac OS X text applications conform to this standard as well. For example, when you create a new multi-line file in TextEdit and save it as plain text, the line endings are LF. This issue is particularly acute in AppleScript when you use `do shell script`, because a multi-line string intended for the Unix shell will usually expect LF as a line-end character.

So, the AppleScript compiler wants your line endings to be CR, but Mac OS X wants your line endings to be LF. The conflict between these forces is always going on. Sometimes this conflict works behind the scenes — for example, the AppleScript compiler *will* change your line endings to CR, no matter what they were before, and that's that. But the conflict also arises up front, every time you press the Return key while you're typing in a script file. At that moment, *some* character needs to be entered, so what should it be?

Script Debugger helps you deal with this conflict, in two main ways:

- **Visibility.** Script Debugger makes it possible for you to see your line endings — you can [show invisibles](#).
- **Freedom of Choice.** Script Debugger lets you choose what character the Return key enters; it's an [Editor preference](#) (New Line Character). The factory default for this preference is LF, and if you create a new multi-line script with the preference set this way, and invisibles showing, you can actually watch AppleScript change the line endings from LF to CR when you compile.

Script Debugger also lets you choose what line endings should be used when you save a script as [text](#). This, however, introduces a further complicating factor.

The problem stems from the fact that in AppleScript, a literal string can span multiple lines:

```
set s to "  
"
```

The integrity of the line-end character within this string (which might be LF or CR) is preserved through decompilation, because behind the scenes the compiled script is just bytecode (and the byte in question is part of a literal string). But now suppose you elect to save this script as text. What should Script Debugger do? The character after the first " is a line ending. If you tell AppleScript to save with CR line endings or LF line endings, this character will become CR or LF, respectively, regardless of what it is "really" supposed to be (thus possibly altering the functionality of your script).

This is why Script Debugger provides the As Is (Mixed) option (the default), which leaves all line endings in the resulting text file the same as in your compiled script. The totally safe solution, though, is not to use any literal line endings within quoted strings. Don't use the "escaped" literals "\r" or "\n", because the AppleScript decompiler turns these into actual line endings. Instead, construct your strings in code, generating line endings at runtime with the `return` global variable or the `ascii character` command. Of course, if you never save as text, then the issue doesn't arise in the first place.

Finally, be aware that pasting (or dragging) from a text file into a script can raise line-end character issues. What is pasted is text, which can have any kind of line ending. Script Debugger does not magically convert these as the paste is performed. If there are multi-line string literals in what is pasted, or if the target script is text, incorrect line endings may now be present in the script. Again, [show invisibles](#) will be a great help here.

◀ [Is Script Debugger's AppleScript the Same as Script Editor's?](#)

[Why Do Applications Open Spontaneously?](#) ▶



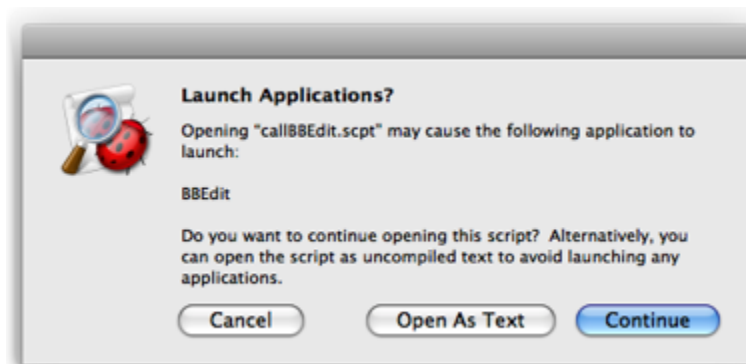
Why Do Applications Open Spontaneously?



Why *do* applications open spontaneously? The trouble is that certain applications must be running in order to provide a [dictionary](#). So every time AppleScript needs the dictionary of such an application — not just in order for a script editor to display a dictionary, but in order to [compile](#) or [decompile](#) a script that targets that application — the application must start up if it isn't running. A very small number of applications have long had "dynamic dictionaries", but with Mac OS X and Cocoa, the proportion of applications that behave this way suddenly went way up. So it's a problem with the system, and with how AppleScript works. It has nothing to do with Script Debugger. In fact, Script Debugger does two things to *reduce* this behavior.

Opening and Decompiling a Script

When you open a script that targets an application which must be launched in order for AppleScript to [decompile](#) it, Script Debugger detects this and [optionally](#) presents a dialog. For example, suppose the script targets BBEdit, which has a dynamic dictionary; you will (if the corresponding [General preference](#) is checked) see this dialog:



You can proceed to open the script (and allow BBEdit to launch) if you wish, but perhaps the overhead of launching an application just to read a script seems unwarranted. If this script was saved with Script Debugger, it contains a [text version](#), and you can click Open As Text to open that instead. Thus you can read the script without launching BBEdit.

To *compile* the script, however, you will have to let AppleScript launch BBEdit. And, if the script does not contain a text version, the Open As Text button won't appear; your only choices will be to open the script and let AppleScript launch BBEdit, or to cancel and not open the script at all.

Opening a Dictionary

Script Debugger makes heavy use of an application's dictionary. For example, in order to calculate the [tell context](#), Script Debugger must load the target application's dictionary. This could cause the target application to launch if it is not running. And Script Debugger needs the tell context when you start to open the File menu (because of the [Open XXX's Dictionary menu item](#)), so there may be a delay as you choose from the File menu, while the target application launches. And then, of course, there's the whole business of what happens when you [search the dictionaries](#) of multiple applications simultaneously.

The good news, however, is that this should happen only once for each application. Script Debugger *caches* an application's dictionary (in `~/Library/Caches/Script Debugger 4.5`) when it opens the dictionary, provided you have not unchecked **Cache generated dictionaries** in the [Dictionary preferences](#). So, as long as the application's dictionary and location don't change, Script Debugger won't have to launch that application again in order to access its dictionary.

Note that this has nothing to do with the discussion under "Opening and Decompiling a Script" earlier on this page. Even when Script Debugger has cached (say) BBEdit's dictionary, AppleScript has not, so when you open a script that targets BBEdit, AppleScript will *still* try to launch BBEdit if it isn't running.

However, you *can* uncheck **Cache generated dictionaries**, or clear the cache on demand by clicking **Clear Cache**, and there are certain specialized circumstances where you might wish to do this. In particular, some applications with 'aete' dictionaries allow those dictionaries to be extended through plug-ins (notable examples are QuarkXPress and InDesign). Script Debugger has no way to notice when you add or remove a plug-in, so the dictionary that it displays, coming from the cached copy, will be out of date.

Applications that do *not* use this plug-in architecture do *not* present any difficulties, and are irrelevant here. If you install a new version of an application, Script Debugger will notice that the dictionary has changed and will automatically refresh the cached copy.

So, if you use these applications, it is up to you to remember to remove the cached copy of the dictionary each time you alter the application by adding or removing plug-ins. There are three ways, or levels, for doing this:

- Uncheck **Cache generated dictionaries**. This is very brute-force, because it prevents caching altogether. Applications will open spontaneously *a lot*, and all opening of dictionaries by Script Debugger will be slower than normal.
 - Keep **Cache generated dictionaries** checked, but click **Clear Cache** when needed. This is only slightly brute-force. You allow caching to work normally, but every once in a while you throw away the caches. So most of the time you are getting all the benefits of caching. But you are throwing away the caches for *all* the dictionaries, when only *one* dictionary (Quark or InDesign) is the problem.
 - Throw away the cache for the problematic application, manually. Quit Script Debugger, open `~/Library/Caches/Script Debugger 4.5`, find the cache for your problem application's dictionary, and move it to the Trash. This is the best solution. The start of the cache file's name will be either the application's name (e.g. "Microsoft Word 7c5d2075.sdef") or its bundle identifier (e.g. "com.apple.mail b521204d.sdef").
-



[What's The Big Deal With Line Endings?](#)

[Hey, Script Debugger Changed My
Formatting!](#)





Hey, Script Debugger Changed My Formatting!



Sometimes it happens that the appearance of your script is changed in ways you never intended. For example, you carefully use the line continuation character to break up a long line of code, like this:

```
set r to display dialog "What is your favorite color?" ¬
    default answer "Blue. No, red!" ¬
    buttons "Aaaaaaaaagh!" ¬
    default button "Aaaaaaaaagh!" ¬
    with title "A Crucial Test"
```

But suddenly it appears broken up all incorrectly, like this:

```
set r to display dialog ¬
    "What is your favorite color?" default answer ¬
    "Blue. No, red!" buttons ¬
    "Aaaaaaaaagh!" default button ¬
    "Aaaaaaaaagh!" with title "A Crucial Test"
```

Why is Script Debugger doing this to you? We're sorry this is happening, but Script Debugger has nothing to do with it. This is a "feature" of AppleScript. It has to do with the fact that a script is [compiled](#) into [bytecode](#) and then displayed to you in decompiled form. There are some annoying behaviors deep within that round-trip process, and this is one of them. There's nothing Script Debugger can do to prevent it.

A related behavior is that although AppleScript will let you *use* a term's synonym, it will also sometimes *replace* the synonym in the decompiled script. So, for example, `close document 2 saving false` is legal, but it is changed to `close document 2 without saving` (and you are probably familiar with the expansion of `app` to `application`, `ref` to a `reference` `to`, and so forth).





Why Doesn't My Script Debug Properly?



There are some known cases where a script that runs fine normally will generate a spurious error when you're in [debug mode](#), thus making it impossible to debug it.

If you run into this sort of situation, try changing any `repeat with ... in blocks` to normal `repeat with loops`. For example, this script chokes in debug mode:

```
tell application "BBEdit"
  tell document 1
    set L to (get every word where its text begins with "t")
    repeat with aWord in L
      tell contents of aWord
        change case of it making capitalize words
      end tell
    end repeat
  end tell
end tell
```

But this version works fine:

```
tell application "BBEdit"
  tell document 1
    set L to (get every word where its text begins with "t")
    repeat with i from 1 to (count L)
      tell (item i of L)
        change case of it making capitalize words
      end tell
    end repeat
  end tell
end tell
```

Let us know if you run into any other cases.



How Do I Script Script Debugger?

Script Debugger 4.5 is scriptable. Moreover, you can script Script Debugger from within Script Debugger itself, either from a script window or from a script in the [Scripts menu](#).

Such a script need not include a tell block targeting Script Debugger; Script Debugger will implicitly be the tell target. However, for development and debugging purposes it is better to supply a tell block (`tell application "Script Debugger 4.5"`), because if you don't, the returned values in the [result pane](#) will be less helpful.

Script Debugger will appear among the applications listed under [File > Open Dictionary](#) and in the [Known Applications inspector](#), so you can easily [open its dictionary](#).

The dictionary is fairly self-explanatory, so a detailed discussion of scripting Script Debugger should be unnecessary. But here are a few points that deserve attention.

Opening Things

Some kinds of Script Debugger [window](#) can be opened:

- To open the Apple Event Log window, say `open Apple Event Log`. You can access the Apple Event log window as the `Apple Event log` property of the application object.
- To open the Scripting Additions dictionary window, say `open scripting additions dictionary`. You can access this window as the `scripting additions dictionary` property of the application object.
- To open an application's dictionary, open the application. The simplest approach is to take advantage of `path to application`; so, for example, `open (path to application "BBedit")`.
- To create a new empty script window, say `make new document`.

All open windows are elements of the application object. You can get a complete list by asking for `every window`, or you can get a list of windows in a particular category by asking for e.g. `every dictionary window` or `every script window`.

Every script window is associated with exactly one document (it has a `document` property). The two classes are not interchangeable, but a script window and its document have the same id. This is fairly standard Cocoa Scripting stuff.

Documents are numbered front to back, so the numbering is volatile. In other words, `document 1` is the document of whatever script window is frontmost at that moment.

Closing Things

To close a window, use the `close` command.

However, when your script running within Script Debugger wants to close a script window (or document), you *must* provide the full form and you must *not* use the `saving ask` option, or Script Debugger will throw an error. So, for example, you can say `close document 2 without saving`, and you can say `close document 2 saving yes` if the file has been previously saved, but you cannot say simply `close document 2`, nor can you say `close document 2 saving ask`.

By the same token, if document 2 has never been saved, you cannot say simply `save document 2`. You must supply the full form, with an `in` parameter specifying where to save.

The reason is that when Script Debugger is scripting itself, we cannot pause the script while the user interacts with the “Do you want to save changes?” dialog or the Save File dialog. So you must avert such interaction by stating explicitly what you want done with the document.

Manipulating Things

As you would expect, most interaction with objects is through their properties. Look, for example, at the properties of a document or a dictionary window or an Apple Event Log window to see all the manipulations you can perform. For example, to switch a script to debug mode, set the document’s `debugger enabled` to `true`.

There are two ways to manipulate the *contents* of a script. First, you can get or set a script’s *entire* contents as text, through the document’s `script source` property; you can also get and set a script’s entire contents as a script object, through the document’s `script` property.

You can also manipulate a script’s *selection*. If you get `selection` you are handed mere text, but if you ask for `selection as point` or `character range of selection`, what you get is a list of two integers, the character offset after the insertion point or the start of the selection (the first character offset is 1) and the count of characters in the selection. Similarly, you can set `selection` to alter the selected text, but if what you set the selection to is a list of two integers, or if you set `character range of selection` to such a list, you reposition the selection.

So, for example, here’s a crude but effective utility for selecting a given line of a given script:

```
on selectLine(num, docnum)
    tell application "Script Debugger 4.5"
        tell document docnum
            set L to every paragraph of (get script source)
            set offs to 1
            repeat with i from 1 to (num - 1)
                set aLine to item i of L
                set offs to offs + (length of aLine) + 1
            end repeat
            set selection to {offs, length of item num of L}
        end tell
    end tell
end selectLine
```

```
        end tell
    end tell
end selectLine
selectLine(3, 2) -- or whatever
```

Running Scripts

Script Debugger provides a repertory of specialized commands for making a script (a document) do such things as compile, execute, step into, and so on.

Execution causes compilation. After calling execute, you can check whether compilation succeeded by examining the document's compiled property.

Some of these commands come with a caveat: execute, pause, step into, step out, and step over all share the feature that they can return before the script has stopped or paused (and start recording can return before the script has started recording). This is a consequence of Script Debugger's multi-threaded architecture.

The correct approach, therefore, if you wish to proceed after the script has done what you asked it to do, is to poll the document's execution state property. So, for example, here's a script that provides a utility for waiting until the document is paused or stopped, and shows how to use it in connection with execute.

```
on waitUntilReady(d)
    repeat while execution state of d is not in {stopped, paused}
        delay 0.2
    end repeat
end waitUntilReady
-- and here's how to use it
tell application "Script Debugger 4.5"
    set d to document "convertFigures"
    tell d
        execute
        my waitUntilReady(d)
        if not compiled then return last error message -- compilation failed, get error
        if last error message is not missing value then -- execution failed, get error
            return last error message
        end if
        return last result -- execution succeeded, get result
    end tell
end tell
```

Future Directions

We'd like to provide fuller scriptability, but this takes work, so we would prefer to go in directions that our users will actually use. If Script Debugger's scriptability lacks a feature you need, do let us know.

◀ [Why Doesn't My Script Debug Properly?](#)