ACME@Kagi.com • wkw@futuris.net

Acme Script Widgets

ACME Script Widgets are a collection of AppleScript™ Scripting Additions which provide powerful text and list manipulation capabilities. AppleScripts will execute from 2 to 20 times faster and require fewer lines of code when using the Widgets.

You may try evaluate ACME Script Widgets for 30 days from the date you first install them. After the evaluation period has expired, you must either purchase a license, or remove them from all computers where they are installed.

Support is available to all ASW customers, as well as people using the Widgets during the evaluation period. Send questions to ACME@kagi.com. The ACME World Wide Web site is currently being built at http://www.gz.com/.

This software may not be uploaded to any online service without written permission.

Ordering Information

To order ACME Script Widgets, use the Register program included with this distribution, or contact Wayne Walrath at 203-857-0631 or by E-mail at ACME@kagi.com. Payment can be made by credit card, check, cash, or with a First Virtual account.

If you are ordering by E-mail, Register disguises your credit card information enough that it isn't obvious what the data is, but it is not guaranteed secure. For E-mail registration, use the "Copy..." button to get the registration information onto the clipboard, then paste it into an E-mail message and send it to shareware@kagi.com. If you are mailing your payment in, use the "Print..." button and a form will be printed which contains your registration information and special bar codes to automate the processing at Kagi. Register also supports printing an invoice which you can give to your purchasing department. To get more help on using Register, turn on Balloon Help after launching the program.

Licenses Fees

Single User License (one CPU)	US$ 10.00
Site License (unlimited copies within same organization in a 100 mile (160 kilometer) radius)	US$ 200.00
World-Wide License (unlimited copies within organization)	US$ 600.00

For commercial bundling, solution provider redistribution or other arrangements, contact Wayne Walrath at acme@kagi.com, (203) 857-0631.

Mail or fax registrations to:

Kagi Shareware
1442-A Walnut Street #392-WW
Berkeley, CA 94709-1405
USA
shareware@kagi.com
fax +1 510 652 6589

Widgets Overview

A short description of each Widget is listed below. To go directly to see the full description of any command, click on its name.

Change Case 1.0

changes the case of text in a half dozen different ways. UPPER, lower, Title, Sentence, and rANdOm case.

Combine Lists 1.0b0

Recombine elements in sublists. Great for manipulating FileMaker Pro records.

Offset In List 1.0.1

Make your own AppleScript™ records (known as association lists or dictionaries). searches a list of items for a specified pattern of characters.

Offsets of 1.1

Get offsets of a string within a string or list of strings.

ACME Parse Args

For parsing and converting CGI form / post arguments. Converts “+” to space, and %XX hex sequences to their corresponding character values, then returns the field names and values as AppleScript™ lists.

ACME Field Lookup

Quickly lookup the value of a field after parsing the CGI form / post arguments. (For use with ACME Parse Args.)

Join List 1.0

Forms a string from a list of items with user specified separators between items. Very flexible way to build strings from a list of items, and easier to use than AppleScript's Text Item Delimiters.

ACME Replace 1.1

Search for text and replace it with other text.

ACME Sort 1.0b4

Sort a list of items or a list of lists. Honors native data types like dates, numbers, et al. Please see the demo scripts for instructions on useage. This was added to late to get into the documentation.

Tokenize 1.1

Split text into a list of items based on specified delimiters.

Trim 1.0b1

Trim unwanted text from either end of a string. Supports the AppleScript™ "ignoring" syntax so you can trim right past other characters or strings.

Mouse 1.0

Get cursor coordinates and button state.

ACME Script Widgets Change History

The ACME Script Widgets are continually being updated and expanded. There were many significant changes and one or two bug fixes between the two public releases (1.0 and 2.0), and most of these are documented below. As the Widgets speed along towards the next major release, several new commands have been added, and the documentation of these commands may not have kept pace with the changes.

Global Changes:

Some of the trivial osaxen, like Balloon Help, have been moved to a sub-folder. While these items are not integral components of the Widgets, they are still being made available. Created single Widget file containing all the commands to simplify installation, but retained the individual Widget files so users could install only the ones they need.

ACME Replace: (formerly called Replace)

Changed the terminology of the osax to reduce the chance of collisions in the terminology space, since many applications define the replace command. Implemented case sensitivity. NOTE: Using the case insensitive option causes Replace to run from a third to half again as long as when it isn't specified, and the memory usage nearly doubles. So don't use this option if you do not need it. Due to the increased memory demands, the osax attempts to allocate space in temporary memory first if it is available, and if not, then attempts to get enough in the local Heap zone.

Offsets Of:

Removed 255 char limit on size of search string. The next version will support passing a list of strings to tokenize instead of having to do it one at a time, and will remove the 255 character limit on the size of delimiter items.

Offset In List:

Totally rewrote the association lists demo and broke it out into its own script. The Association List is now a script object with methods for all the common operations. You can load it into your scripts as a property and get flexible association lists effortlessly.

Mouse:

New addition to satisfy a user's request. Donald Olsen was the first to create this command I believe, and this version doesn't do anything that his doesn't.

Combine Lists:

New to version 2.0. Useful for working lists of records and repeating fields from FileMaker Pro.

Trim:

New to version 2.0. Trims any number of strings or characters off either end of the target strings, with the option of ignoring any number of patterns. This is going to be one of the hot-ones!!

Tokenize:

Version 1.2 was rewritten to return null tokens.

Installation

The Scripting Additions must be copied to the Scripting Additions folder which resides inside the Extensions folder of the System folder on English language versions of the Macintosh operating system. For non-English versions, please refer to the documentation which came with the operating system if you are unsure where they belong.

Installing ACME Script Widgets:

1.) The file ACME Script Widgets 2.5 contains all of the core Widgets. Other special purpose Widgets are provided in separate files so they only need to be installed if required.

ACME Replace

ACME Replace is a general purpose search and replace tool for AppleScript™. You can specify one or more strings to search for in one or more target strings, and each occurrence will be replaced by the specified replacement text. Replace has been renamed to ACME Replace to avoid terminology conflicts with other Scripting Additions and applications.

USAGE:

```
ACME replace: replaces occurrences of a string with something else.
ACME replace  anything  -- string or list of strings to search for.
        in  anything  -- the string or list of strings to search in.
        with  string  -- the replace string.
        [all occurrences  boolean]  -- replace all occurrences?  (default=TRUE)
        [case  sensitive/insensitive]  -- Consider case? (Default is sensitive.)

Result:   anything  -- original text item(s) with replacements.
```

There are no limits on the size of the parameters (other than available memory), and anything that AppleScript™ can coerce to a string is legal (e.g., integers, floats, etc.).

If you pass a list of strings to search for, they are replaced one at a time in the order they are passed. If you specify that only the first occurrence should be replaced, each one of the search strings will be replaced once. This might be useful if you wanted to replace the first X number of a certain string. For example:

```
ACME replace {1, 1, 1} in "12121212" with "_" without all occurrences

=>      "_2_2_212"
```

Three of the ones (1) were replaced with "_", and the rest left untouched.

NOTE: Using the case insensitive option causes Replace to run slower than when it isn't specified, and the memory usage nearly doubles. So don't use this option if you do not need it. Due to the increased memory demands when using the case insensitive parameter, the osax attempts to allocate space in temporary memory first if it is available, and if not, then attempts to get enough in the local Heap zone.

VERSION HISTORY:

vers 1.1 3Mar95
-Implemented "case" parameter and changed how its terminology.
-Renamed it to ACME Replace to avoid terminology conflicts.

Change Case

Change Case transforms the case of the text passed to it in a variety of ways. This Scripting Addition is most likely ONLY useful for working with English language text, since other languages have their own rules for capitalization.

USAGE:

change case: changes the case of (Roman) text.

change case
    of anything -- text to modify. (string or list)
    [to upper/lower/title case/sentence case/toggle case/who cares]

Result: anything -- original text with case changed.

The upper and lower case transforms do as their name implies.

Title case makes the first alpha character after a non-alpha character upper case.

Sentence case searches for the first alpha character in the string and capitalizes it, then continues on searching for one of /! / . / ? / and capitalizes the next alpha character after seeing one of the punctuation marks (all other characters are left untouched).

Toggle just switches the case of each character from whatever it was.

The "who cares" option was added mostly for fun (well, totally for fun) and randomly changes the case of each letter (if anyone finds a use for this option other than creating electronic ransom notes, please let me know...).

Combine Lists

Combine Lists reorganizes the items in sublists, grouping the Nth item from each of the inner lists together into a new list. For instance, the list {{1,2},{a,b}} becomes {{1,a},{2,b}} after processing. This Widget is quite handy to have around when scripting FileMaker Pro. When FileMaker Pro returns multiple records, they come back as a list of lists. If you send this list through Combine Lists, like fields from each record will be grouped together in a list. The idea for this Widget came from Dennis Whiteman, The Ultimate Freelancer.

USAGE:

combine lists: Groups Nth items from sublists together.
    combine lists list -- List of lists.
    Result: list -- Returns list of lists with items reorganized.

given the following list:
{ {"a","the","what a"}, {"dead", "fat", "huge"},{"cat","dog","bird"} }

Combine Lists returns:
{{"a", "dead", "cat"}, {"the", "fat", "dog"}, {"what a", "huge", "bird"}}

The Nth item from each sublist is grouped together into a new sublist.

Running the output back through Combine Lists a second time gives you the original list back. Combine Lists requires that all sublists have the same number of elements or it returns an error.

FileMaker Pro Users
When scripting Claris' FileMaker Pro, if you ask for a range of records, they will be returned as a list of lists with each inner list containing the fields from a single record. This Widget, when applied to a list of records will group the data from the individual fields together in the inner lists.

Join List

Join List forms a string from a list of items, inserting delimiters (in a repeating fashion if their are more than one) between the strings.

USAGE:

join list: form string from list items with the delimiters inserted in between.

```
join list  list  -- list of text items to join.
        with delimiters  list  -- list of delimiter items.

Result:   string  -- Returns the joined list as a string.
```

The first parameter is a list of zero or more strings to be joined into a string. There is no limit on the size of the items other than available memory.

The second parameter specifies one or more strings to insert between the joined items.

Delimiters are inserted between items of the first list sequentially; when the end of the delimiter list is reached, the Scripting Addition begins again with the first delimiter (they are inserted in a rotating fashion).

EXAMPLES:

```
join list {"Join", "List", "Ver1.0"} with delimiters {space}
        => "Join List Ver1.0"
```

The _space_ character is repeatedly inserted between the items in the first list. I don't think this needs much more explaining, so let's move on to more complex examples.

```
set jList to {"One", "Two", "Three", "Four", "Five"}
set dList to {"$", "#"}
join list jList with delimiters dList
        => "One$Two#Three$Four#Five"
```

The "$" is inserted between the first two items and the "#" between the second and third items, then because the end of the delimiter list was reached, it starts over at the beginning of the delimiters again with "$", continuing on in this manner until all of the strings in jList have been added.

If an empty list of delimiters is specified ({}), the command behaves exactly as if you had used AppleScript™ to coerce the list of strings to a single string.

```
set jList to {"One", "Two", "Three", "Four", "Five"}
join list jList with delimiters {}
        => "OneTwoThreeFourFive"
```

If the list of strings to be joined contains only a single element, join list returns just that element with none of the tokens appended.

```
join list "one" with delimiters {"&", "*"}
        => "one"
```

If you want any of the delimiters on the front or end of the returned string, use AppleScript's built-in capabilities for this.

```
set jList to {"One", "Two", "Three", "Four"}
set myString to "|" & (join list jList with delimiters "|") & "|"
        => "|One|Two|Three|Four|"
```

VERSION HISTORY:

* ver 1.1 --15Nov94
- Removed limit on size of delimiter items.
- Removed the syntax for choosing where to insert delimiters and the optional ending parameter.

* ver1.0 -- 2Nov94
- First public release.

Offset In List

Offset In List searches through a list of items for a string. It searches for the pattern in each item of the list, or optionally only even or odd numbered items, and there are options for performing exact match and case sensitive searches.

USAGE:

offset in list: search list items, returning found string's index or next item.

offset in list list -- list of items to search.
    of string -- the search string.
    [returning next item boolean] -- (index returned by default)
    [searching even items/odd items/all items] -- Default: check every item.
    [case sensitive/insensitive] -- ignore case? (Default: case is significant.)
    [exact match boolean] -- whole word matching only? (Default = true.)

Result: anything -- item number of target, or (optionally) the next item after the target.

Considering the parameters in order, here's what they are for:

The direct parameter <list> is the list of items to search through.

<string> is the target to search for.

Returning... determines whether you want the item number where the target is found to be returned, or instead the next item after the item where target is found. Returning the offset is the default. The intended use for this option is in working with association lists (poor-man's records), where you have <key, value> pairs in a list.

Searching ... determines whether every item is searched or only even or odd items. If you are working with <key, value> pairs, you would only want to search odd items to avoid finding find the target in a value item instead of in a key item. Default is to search every item.

Case ... determines whether case is considered in the search. Sensitive is default.

Exact match... if this is set to true (default) the match must be exact (however, using the _case_ option modifies this behavior), if set to false, it will look for substring matches.

The primary use for Offset In List is to easily work with association lists. Association lists are key/value pairs which function more or less like AppleScript™ records. With Offset In List, you can store key/value pairs in a regular AppleScript™ list then use Offset In List to search the keys and return the value for a matched key. Consider the following contrived example of a list of names and phone numbers.

```
set people to {"Stephan", "455-1234", "Renee", "455-4444", "Laura", "433-2345"}
```

When we want to lookup a person's number, we search the keys in the list (the names). Notice that the keys are the odd numbered items of the list: 1, 3, and 5. Here's how to do this with Offset In List:

```
offset in list people of "renee" case insensitive searching odd items with returning next item
```

This statement would locate the key "Renee", and return the next item in the list which is "455-4444". We tell Offset In List to search only odd numbered items, and to ignore the case of the key.

It's a little more elegant to use AppleScript's records for this task, but unfortunately, with AppleScript™ you have to define the record and labels at the script's compile time. Using Offset In List, you don't need to.

An entire Association List script object has been provided in the demos. It contains operations for adding, deleting, and looking up data, plus several more.

EXAMPLES:

```
offset in list {"Apple", "Script"} of "apple"
    => 0          -- not found because case sensitive search performed by default
offset in list {"Apple", "Script"} of "apple" case insensitive
   => 1           -- found in first element of list
offset in list {"Apple", "Script"} of "Scr" without exact match
   => 2           -- matched first three letters of second item
offset in list {"Apple", "Script"} of "App" with returning next item without exact match
    => "Script"           -- matched first item and returned next item
offset in list {"Apple", "Script"} of "App" searching even items with returning next item
without exact match
   => ""          -- didn't match the first item because only even items searched
```

VERSION HISTORY:

vers 1.0.1 --15Feb95
Fixed terminology conflict which kept AppleScript™ from recognizing the form "every item of X".
The _Searching_ optional parameter now uses "all items" instead of "every item."
Added Association Lists demo script.

Offsets Of

Offsets Of searches through a string or list of strings for a search pattern and returns a list of offsets (indexes into the string) where the string was found.

USAGE:

offsets of: returns a list of offsets of one string in another.

offsets of string -- Search string.
in anything -- string or list of strings to search.
[case sensitive/insensitive] -- defaults to case sensitive.

Result: list -- list of offsets where search string was found.

If the case parameter is not specified it defaults to sensitive; that is, case IS significant, so "and" is not the same as "AND".

If you pass a single string to search in, the result is a list of the offsets, or an empty list if no matches were found. If you pass a list of strings to be searched, a list of lists of offsets is returned, with some of the lists possibly empty if no matches were made. There are no size limits on the parameters other than available memory.

Tokenize

Tokenize was designed to make it easier to split text into elements based on a set of delimiters. The demo AppleScript™ illustrates several novel uses for Tokenize which may not be obvious at first glance.

USAGE:

```
tokenize: split text into a list of items based on list of delimiters.

tokenize  string  -- the string to tokenize.
        with delimiters  anything  -- the delimiter string(s).

Result:   list  -- list of token strings.
```

the direct parameter to tokenize is a string, and the second (required) parameter is a list of strings (each string being one or more bytes in length) to use in tokenizing the direct parameter.

If you are only tokenizing with one delimiter you need not pass it as a list since AppleScript™ will handle the coercion for you. For example, the following is legal:

```
tokenize "My Name Is" with delimiters " Name "
        => {"My", "Is"}
```

What are tokens? Tokens can be anything which has some particular meaning within a context. The words in this sentence can be considered tokens. Each word has some meaning in the English language. The spaces between the words have no special meaning (for this discussion) except to delimit where the tokens start and stop. If the first sentence of this paragraph is run through Tokenize with a delimiter list consisting of a single space character, and the punctuation mark "?", it would return a list of three items (words): {"What", "are", "tokens"}. This is the process of tokenization.

Here's what that would look like in AppleScript™:

```
tokenize "What are tokens?" with delimiters {space, "?"}
        => {"What", "are", "tokens"}
```

Tokenize lets you specify which patterns to use as token delimiters, then it searches through a piece of text pulling out all the sequences of characters found between the specified tokens. ("space" is an AppleScript™ constant which translates to the space character [' ' or ASCII 32]).

Some text processing tasks require more than one call to Tokenize to perform. For example, if the variable myText contained a number of lines separated by

return characters, and you wanted to retrieve the words from line five, you could write the following AppleScript™ commands:

```
tokenize myText with delimiters {return}
tokenize (item 5 of result) with delimiters {space}
        => [result is a list with all the words from line five of the text]
```

BACKGROUND INFORMATION:

Because of the way the tokenization is implemented, Tokenize can also be used as a quick way of removing unwanted characters from a text string. To better understand what is possible with Tokenize, here's a brief description of how Tokenize functions. The text to be tokenized is scanned for each of the strings given in the delimiter list, and all occurrences of these strings are replaced by a special character (essentially a null-char). After all delimiters are processed, a final pass is made which gathers all the strings between the special characters into a list. Understanding this algorithm will help you to figure out how text will ultimately be parsed when using Tokenize.

For example, consider an arbitrary string of text which contains words separated by tab characters, and between each word there will be one to three tabs. Here's a string set up as described:

```
set testString to "One\tGiant\t\tStep\tFor\t\t\tMankind"    -- NOTE tabs ("\t"
```

Using tab as the only delimiter returns the following:

```
tokenize testString with delimiters tab
        => {"One", "Giant", "Step", "For", "Mankind"}
```

If, on the other hand, tokenize is run using a string of three tabs, the output is different:

```
tokenize testString with delimiters tab & tab & tab
        =>      {"One   Giant           Step    For", "Mankind"}
```

The output from this version consists of a list of two strings. Since tokenize only found one place in the testString where there were three tab characters side by side, it split the string there. Tokenizing with a two tab string would produce yet a different result.

VERSION HISTORY:

ver 1.2b1 26Jun95
Rewritten to return null tokens. Now the behavior exactly matches that of AppleScript when using the text item delimiters.

ver 1.1 - 2Nov94
Fixed bug which surfaced in ver 1.0 when the tokens were longer than 255 chars. This bug resulted in random memory being stomped on when the token was too long. All users of version 1.0 should switch immediately. Cleaned the code up and optimized it a bit.

ver 1.0 - Oct94
First release.

Trim

Trim removes unwanted characters or patterns from the beginning and end of strings, optionally ignoring certain patterns.

USAGE:

```
trim: trim patterns of text off the ends of strings.

trim  [anything]  -- the patterns to trim. (string or list)
        off  anything  -- the items to trim.
        [from  front side/back side/both sides]  -- location to trim from.
        [ignoring  anything]  -- string(s) to ignore.

Result:  anything  -- the trimmed text.
```

Trim accepts a string or list of strings to be trimmed, a string or list of strings specifying what to trim, and optionally lets you specify where to trim from (from the front, back or both sides of the strings), and which pattern(s) should be ignored (a string or list of strings).

There are no size limits on the size of the parameters (except for available memory). If no items are specified for trimming, Trim will remove spaces from the end of the text items, ignoring nothing. Likewise, if the from parameter is not specified, only the end of the string(s) will be trimmed.

To summarize Trim's defaults, only the second parameter is required (the stuff to trim), and Trim will remove single spaces from only the end of the strings, ignoring nothing.

The results of specifying the same pattern as a trim argument and also an ignoring argument is indeterminate.

EXAMPLES:

```
Trim {space} off "    On a clear day you can see forever...    "
    => "    On a clear day you can see forever..."
```

The extra spaces are removed from the end of the string. If however, the string ends with a return character, then we should use the following form:

```
Trim {space} off "  On a clear day you can see forever...    " & return
    => "  On a clear day you can see forever...\r"   -- "\r" is the return character
```

The spaces at the front of the string are left alone because default is to only trim from the end of the string. We can nab those spaces as well by specifying the from parameter.

```
Trim {space} off "  On a clear day you can see forever...    " & return from both sides
    => "On a clear day you can see forever...\r"   ---- "\r" is the return character
```

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY. THIS PROGRAM IS SOLD WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES WHATSOEVER. BECAUSE OF THE DIVERSITY OF CONDITIONS AND HARDWARE UNDER WHICH THIS PROGRAM MAY BE USED, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. THE USER MUST ASSUME THE ENTIRE RISK OF USING THE PROGRAM. ANY LIABILITY OF SELLER OR AUTHOR WILL BE LIMITED EXCLUSIVELY TO REPLACEMENT OR REFUND OF THE PURCHASE PRICE.

Government End Users: If you are acquiring the Software on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Software is supplied to the Department of Defense (DoD), the Software is classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Software, and its documentation as that term is defined in Clause 252.227-7013(c)(1) of the DFARS;

and (ii) if the Software is supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Software, and its documentation will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in Clause 18-52.227-86(d) of the NASA Supplement to the FAR.

ACME Parse Args
ACME Lookup Field

ACME Parse Args and Field Lookup handle the tedious task of converting and parsing the arguments passed to a CGI script. The Parse command converts all "+" characters to a space, and translates sequences of %XX (where XX are hex digits) to the proper character. Additionally, Parse will combine the values for all duplicate field names into a single list.

The ACME Field Lookup command provides a very fast way to access specific field values in the parsed argument list.

Both of these commands were still undergoing changes as the documentation was prepared, so no further information is available here. Please see the demo scripts and README files included in the package or contact us at ACME@kagi.com.