

master ▾

...

zsh-completions / zsh-completions-howto.org

 **syohex** Update links

 History

7 contributors



 464 lines (414 sloc) | 28.9 KB

...

Table of Contents

- [Intro](#)
- [Getting started](#)
 - [Telling zsh which function to use for completing a command](#)
 - [Completing generic gnu commands](#)
 - [Copying completions from another command](#)
- [Writing your own completion functions](#)
 - [Utility functions](#)
 - [Writing simple completion functions using `_describe`](#)
 - [Writing completion functions using `_alternative`](#)
 - [Writing completion functions using `_arguments`](#)
 - [Writing completion functions using `_regex_arguments` and `_regex_words`](#)
 - [complex completions with `_values`, `_sep_parts`, & `_multi_parts`](#)
 - [Adding completion words directly using `compadd`](#)
- [Testing & debugging](#)
- [Gotchas \(things to watch out for\)](#)
- [Tips](#)
- [Other resources](#)

Intro

The official documentation for writing zsh completion functions is difficult to understand, and doesn't give many examples. At the time of writing this document I was able to find a few other tutorials on the web, however those tutorials only explain a small portion of the capabilities of the completion system. This document aims to cover areas not explained elsewhere, with examples, so that you can learn how to write more advanced completion functions. I do not go into all the details, but will give enough information and examples to get you up and running. If you need more details you can look it up for yourself in the [official documentation](#).

Please make any scripts that you create publicly available for others (e.g. by forking this repo and making a pull request). Also if you have any more information to add or improvements to make to this tutorial, please do.

Getting started

Telling zsh which function to use for completing a command

Completion functions for commands are stored in files with names beginning with an underscore `_`, and these files should be placed in a directory listed in the `$fpath` variable. You can add a directory to `$fpath` by adding a line like this to your `~/.zshrc` file:

```
fpath=(~/newdir $fpath)
```

The first line of a completion function file can look something like this:

```
#compdef foobar
```

This tells zsh that the file contains code for completing the `foobar` command. This is the format that you will use most often for the first line, but you can also use the same file for completing several different functions if you want. See [here](#) for more details.

You can also use the `compdef` command directly (e.g. in your `~/.zshrc` file) to tell zsh which function to use for completing a command like this:

```
> compdef _function foobar
```

or to use the same completions for several commands:

```
> compdef _function foobar goocar hoodar
```

or if you want to supply arguments:

```
> compdef '_function arg1 arg2' foobar
```

See [here](#) for more details.

Completing generic gnu commands

Many [gnu](#) commands have a standardized way of listing option descriptions (when the `--help` option is used). For these commands you can use the `_gnu_generic` function for automatically creating completions, like this:

```
> compdef _gnu_generic foobar
```

or to use `_gnu_generic` with several different commands:

```
> compdef _gnu_generic foobar goocar hoodar
```

This line can be placed in your `~/.zshrc` file.

Copying completions from another command

If you want a command, say `cmd1`, to have the same completions as another, say `cmd2`, which has already had completions defined for it, you can do this:

```
> compdef cmd1=cmd2
```

This can be useful for example if you have created an alias for a command to help you remember it.

Writing your own completion functions

A good way to get started is to look at some already defined completion functions. On my linux installation these are found in `/usr/share/zsh/functions/Completion/Unix` and `/usr/share/zsh/functions/Completion/Linux` and a few other subdirs.

You will notice that the `_arguments` function is used a lot in these files. This is a utility function that makes it easy to write simple completion functions. The `_arguments` function is a wrapper around the `compadd` builtin function. The `compadd` builtin is the core function used to add completion words to the command line, and control its behaviour. However, most of the time you will not need to use `compadd`, since there are many utility functions such as `_arguments` and `_describe` which are easier to use.

For very basic completions the `_describe` function should be adequate

Utility functions

Here is a list of some of the utility functions that may be of use. The full list of utility functions, with full explanations, is available [here](#). Examples of how to use these functions are given in the next section.

main utility functions for overall completion

<code>_alternative</code>	Can be used to generate completion candidates from other utility functions or shell code.
<code>_arguments</code>	Used to specify how to complete individual options & arguments for a command with unix style options.
<code>_describe</code>	Used for creating simple completions consisting of words with descriptions (but no actions). Easier to use than <code>_arguments</code>
<code>_gnu_generic</code>	Can be used to complete options for commands that understand the <code>--help</code> option.
<code>_regex_arguments</code>	Creates a function for matching commandline arguments with regular expressions, and then performing actions/completions.

functions for performing complex completions of single words

<code>_values</code>	Used for completing arbitrary keywords (values) and their arguments,
----------------------	--

	or comma separated lists of such combinations.
<code>_combination</code>	Used to complete combinations of values, for example pairs of hostnames and usernames.
<code>_multi_parts</code>	Used for completing multiple parts of words separately where each part is separated by some char, e.g. for completing partial filepaths: <code>/u/i/sy -> /usr/include/sys</code>
<code>_sep_parts</code>	Like <code>_multi_parts</code> but allows different separators at different parts of the completion.
<code>_sequence</code>	Used as a wrapper around another completion function to complete a delimited list of matches generated by that other function.

functions for completing specific types of objects

<code>_path_files</code>	Used to complete filepaths. Take several options to control behaviour.
<code>_files</code>	Calls <code>_path_files</code> with all options except <code>-g</code> and <code>-/</code> . These options depend on <code>file-patterns</code> style setting.
<code>_net_interfaces</code>	Used for completing network interface names
<code>_users</code>	Used for completing user names
<code>_groups</code>	Used for completing group names
<code>_options</code>	Used for completing the names of shell options.
<code>_parameters</code>	Used for completing the names of shell parameters/variables (can restrict to those matching a pattern).

functions for handling cached completions

If you have a very large number of completions you can save them in a cache file so that the completions load quickly.

<code>_cache_invalid</code>	indicates whether the completions cache corresponding to a given cache identifier needs rebuilding
<code>_retrieve_cache</code>	retrieves completion information from a cache file
<code>_store_cache</code>	store completions corresponding to a given cache identifier in a cache file

other functions

<code>_message</code>	Used for displaying help messages in places where no completions can be generated.
<code>_regex_words</code>	Can be used to generate arguments for the <code>_regex_arguments</code> command. This is easier than writing the arguments manually.
<code>_guard</code>	Can be used in the ACTION of specifications for <code>_arguments</code> and similar functions to check the word being completed.

Actions

Many of the utility functions such as `_arguments`, `_regex_arguments`, `_alternative` and `_values` may include an action at the end of an option/argument specification. This action indicates how to complete the corresponding argument. The actions can take one of the following forms:

<code>()</code>	Argument is required but no matches are generated for it.
<code>(ITEM1 ITEM2)</code>	List of possible matches
<code>((ITEM1\:'DESC1' ITEM2\:'DESC2'))</code>	List of possible matches, with descriptions. Make sure to use different quotes than those around the whole specification.
<code>->STRING</code>	Set <code>\$state</code> to <code>STRING</code> and continue (<code>\$state</code> can be checked in a case statement after the utility function call)
<code>FUNCTION</code>	Name of a function to call for generating matches or performing some other action, e.g. <code>_files</code> or <code>_message</code>
<code>{EVAL-STRING}</code>	Evaluate string as shell code to generate matches. This can be used to call a utility function with arguments, e.g. <code>_values</code> or <code>_describe</code>
<code>=ACTION</code>	Inserts a dummy word into completion command line without changing the point at which completion takes place.

Not all action types are available for all utility functions that use them. For example the `->STRING` type is not available in the `_regex_arguments` or `_alternative` functions.

Writing simple completion functions using `_describe`

The `_describe` function can be used for simple completions where the order and position of the options/arguments is not important. You just need to create an array parameter to hold the options & their descriptions, and then pass the parameter name as an argument to `_describe`. The following example creates completion candidates `c` and `d`, with the descriptions (note this should be put in a file called `_cmd` in some directory listed in `$fpath`).

```
#compdef cmd
local -a subcmds
subcmds=('c:description for c command' 'd:description for d command')
_describe 'command' subcmds
```

You can use several different lists separated by a double hyphen as follows but note that this mixes the matches under and single heading and is not intended to be used with different types of completion candidates:

```
local -a subcmds topics
subcmds=('c:description for c command' 'd:description for d command')
topics=('e:description for e help topic' 'f:description for f help topic')
_describe 'command' subcmds -- topics
```

If two candidates have the same description, `_describe` collects them together on the same row and ensures that descriptions are aligned neatly in columns. The `_describe` function can be used in an ACTION as part of a specification for `_alternative`, `_arguments` or `_regex_arguments`. In this case you will have to put it in braces with its arguments, e.g. `'TAG:DESCRIPTION:{_describe 'values' options}'`

Writing completion functions using `_alternative`

Like `_describe`, this function performs simple completions where the order and position of options/arguments is not important. However, unlike `_describe`, instead of fixed matches further functions may be called to generate the completion candidates. Furthermore, `_alternative` allows a mix of different types of completion candidates to be mixed.

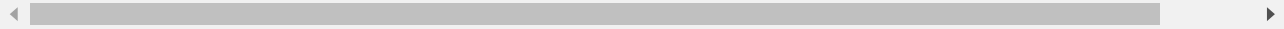
As arguments it takes a list of specifications each in the form `'TAG:DESCRIPTION:ACTION'` where TAG is a special tag that identifies the type of completion matches, DESCRIPTION is used as a heading to describe the group of completion candidates collectively, and ACTION is one of the action types listed previously (apart from the `->STRING` and `=ACTION` forms). For example:

```
_alternative 'arguments:custom arg:(a b c)' 'files:filename:_files'
```

The first specification adds completion candidates a, b & c, and the second specification calls the `_files` function for completing filepaths.


We could split the specifications over several lines with `\` and add descriptions to each of the custom args like this:

```
_alternative \  
  'args:custom arg:((a\: "description a" b\: "description b" c\: "description c" \  
  'files:filename:_files'
```



If we want to pass arguments to `_files` they can simply be included, like this:

```
_alternative \  
  'args:custom arg:((a\: "description a" b\: "description b" c\: "description c" \  
  'files:filename:_files -/'
```



To use parameter expansion to create our list of completions we must use double quotes to quote the specifications, e.g:

```
_alternative \  
  "dirs:user directory:($userdirs)" \  
  "pids:process ID:($(ps -A o pid=))"
```

In this case the first specification adds the words stored in the `$userdirs` variable, and the second specification evaluates `'ps -A o pid='` to get a list of pids to use as completion candidates. In practice, we would make use of the existing `_pids` function for this.

We can use other utility functions such as `_values` in the ACTION to perform more complex completions, e.g:

```
_alternative \  
  "directories:user directory:($userdirs)" \  
  'options:comma-separated opt: _values -s , letter a b c'
```

this will complete the items in `$userdirs`, as well as a comma separated list containing a, b &/or c. Note the use of the initial space before `_values`. This is needed because `_values` doesn't understand standard compadd options for descriptions.

As with `_describe`, the `_alternative` function can itself be used in an ACTION as part of a specification for `_arguments` or `_regex_arguments`.

Writing completion functions using `_arguments`

With a single call to the `_arguments` function you can create fairly sophisticated completion functions. It is intended to handle typical commands that take a variety of options along with some normal arguments. Like the `_alternative` function, `_arguments` takes a list of specification strings as arguments. These specification strings specify options and any corresponding option arguments (e.g. `-f filename`), or command arguments.

Basic option specifications take the form `'-OPT[DESCRIPTION]'`, e.g. like this:

```
_arguments '-s[sort output]' '--l[long output]' '-l[long output]'
```

Arguments for the option can be specified after the option description in this form `'-OPT[DESCRIPTION]:MESSAGE:ACTION'`, where MESSAGE is a message to display and ACTION can be any of the forms mentioned in the ACTIONS section above. For example:

```
_arguments '-f[input file]:filename:_files'
```

Command argument specifications take the form `'N:MESSAGE:ACTION'` where N indicates that it is the Nth command argument, and MESSAGE & ACTION are as before. If the N is omitted then it just means the next command argument (after any that have already been specified). If a double colon is used at the start (after N) then the argument is optional. For example:

```
_arguments '-s[sort output]' '1:first arg:_net_interfaces' '::optional arg:_f
```

here the first arg is a network interface, the next optional arg is a file name, the last arg can be either a, b or c, and the `-s` option may be completed at any position.

The `_arguments` function allows the full set of ACTION forms listed in the ACTION section above. This means that you can use actions for selecting case statement branches like this:

```
_arguments '-m[music file]:filename:->files' '-f[flags]:flag:->flags'
case "$state" in
    files)
```

```

    local -a music_files
    music_files=( Music/**/*.{mp3,wav,flac,ogg} )
    _multi_parts / music_files
    ;;
flags)
    _values -s , 'flags' a b c d e
    ;;
esac

```

In this case paths to music files are completed stepwise descending down directories using the `_multi_parts` function, and the flags are completed as a comma separated list using the `_values` function.

I have just given you the basics of `_arguments` specifications here, you can also specify mutually exclusive options, repeated options & arguments, options beginning with + instead of -, etc. For more details see the [official documentation](#). Also have a look at the tutorials mentioned at the end of this document, and the completion functions in the [src directory](#).

Writing completion functions using `_regex_arguments` and `_regex_words`

If you have a complex command line specification with several different possible argument sequences then the `_regex_arguments` function may be what you need. It typically works well where you have a series of keywords followed by a variable number of arguments.

`_regex_arguments` creates a completion function whose name is given by the first argument. Hence you need to first call `_regex_arguments` to create the completion function, and then call that function, e.g. like this:

```

_regex_arguments _cmd OTHER_ARGS .
_cmd "$@"

```

The `OTHER_ARGS` should be sequences of specifications for matching & completing words on the command line. These sequences can be separated by `|` to represent alternative sequences of words. You can use bracketing to arbitrary depth to specify alternate subsequences, but the brackets must be backslashed like this `\(\)` or quoted like this `(' ')`.

For example:

```
_regex_arguments _cmd SEQ1 '|' SEQ2 \( SEQ2a '|' SEQ2b \)
_cmd "$@"
```

This specifies a command line matching either SEQ1, or SEQ2 followed by SEQ2a or SEQ2b. You are describing the form arguments to the command take in the form of a regular expression grammar.

Each specification in a sequence must contain a / PATTERN/ part at the start followed by an optional ':TAG:DESCRIPTION:ACTION' part.

Each PATTERN is a regular expression to match a word on the command line. These patterns are processed sequentially until we reach a pattern that doesn't match at which point any corresponding ACTION is performed to obtain completions for that word. Note that there needs to be a pattern to match the initial command itself. See below for further explanation about PATTERNS.

The ':TAG:DESCRIPTION:ACTION' part is interpreted in the same way as for the `_alternative` function specifications, except that it has an extra : at the start, and now all of the possible ACTION formats listed previously are allowed.

Here is an example:

```
_regex_arguments _cmd /$'[^\\0]##\\0'/ \( /$'word1(a|b|c)\\0'/ ':word:first word
/$'word11(a|b|c)\\0'/ ':word:first word:(word11a word11b word11c)' \( /$'wo
'|' /$'word22(a|b|c)\\0'/ ':word:second word:(word22a word22b word22c)' \)
_cmd "$@"
```

in this case the first word can be word1 or word11 followed by an a, b or c, and if the first word contains 11 then a second word is allowed which can be word2 followed by and a, b, or c, or a filename.

If this sounds too complicated a much simpler alternative is to use the `_regex_words` function for creating specifications for `_regex_arguments`.

Patterns

You may notice that the / PATTERN/ specs in the previous example don't look like normal regular expressions. Often a string parameter in the form \$'foo\0' is used. This is so that the \0 in the string is interpreted correctly as a null char which is used to separate words in the internal representation. If you don't include the \0 at the end of the pattern you may get problems matching the next word. If you need to use the contents of a variable in a pattern, you can double quote it so that it gets expanded and then put a string parameter containing a null char afterwards, like this: "\$somevar"\$'\0'

The regular expression syntax for patterns seems to be a bit different from normal regular expressions, and I can't find documentation anywhere. However I have managed to work out what the following special chars are for:

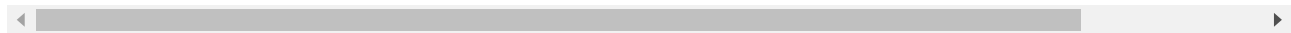
*	wildcard - any number of chars
?	wildcard - single char
#	zero or more of the previous char (like * in a normal regular expression)
##	one or more of the previous char (like + in a normal regular expression)

_regex_words

The `_regex_words` function makes it much easier to create specifications for `_regex_arguments`. The results of calling `_regex_words` can be stored in a variable which can then be used instead of a specification for `_regex_arguments`.

To create a specification using `_regex_words` you supply it with a tag followed by a description followed by a list of specifications for individual words. These specifications take the form 'WORD:DESCRIPTION:SPEC' where WORD is the word to be completed, DESCRIPTION is a description for it, and SPEC can be another variable created by `_regex_words` specifying words that come after the current word or blank if there are no further words. For example:

```
_regex_words firstword 'The first word' 'word1a:a word:' 'word1b:b word:' 'wo
```



the results of this function call will be stored in the \$reply array, and so we should store it in another array before \$reply gets changed again, like this:

```
local -a firstword
_regex_words word 'The first word' 'word1a:a word:' 'word1b:b word:' 'word1c:c
```

```
firstword="$reply[@]"
```

we could then use it with `_regex_arguments` like this:

```
_regex_arguments _cmd /$'[^\\0]##\\0'/ "$firstword[@]"  
_cmd "$@"
```

Note that I have added an extra pattern for the initial command word itself.

Here is a more complex example where we call `_regex_words` for different words on the command line

```
local -a firstword firstword2 secondword secondword2  
_regex_words word1 'The second word' 'woo:tang clan' 'hoo:not me'  
secondword=("$reply[@]")  
_regex_words word2 'Another second word' 'yee:thou' 'haa:very funny!'  
secondword2=("$reply[@]")  
_regex_words commands 'The first word' 'foo:do foo' 'man:yeah man' 'chu:at chu'  
firstword=("$reply[@]")  
_regex_words word4 'Another first word' 'boo:scare somebody:$secondword' 'ga:|  
'loo:go to the toilet:$secondword2'  
firstword2=("$reply[@]")  
  
_regex_arguments _hello /$'[^\\0]##\\0'/ "${firstword[@]}" "${firstword2[@]}"  
_hello "$@"
```

In this case the first word can be one of “foo”, “man”, “chu”, “boo”, “ga” or “loo”. If the first word is “boo” or “ga” then the second word can be “woo” or “hoo”, and if the first word is “loo” then the second word can be “yee” or “haa”, in the other cases there is no second word.

For a good example of the usage of `_regex_words` have a look at the `_ip` function.

complex completions with `_values`, `_sep_parts`, & `_multi_parts`

The `_values`, `_sep_parts` & `_multi_parts` functions can be used either on their own, or as ACTIONS in specifications for `_alternative`, `_arguments` or `_regex_arguments`. The following examples may be instructive. See the [official documentation](#) for more info.

Space separated list of mp3 files:

```
_values 'mp3 files' ~/.*.mp3
```

Comma separated list of session id numbers:

```
_values -s , 'session id' "${(uonzf)}$(ps -A o sid=)}"
```

Completes foo@news:woo, or foo@news:laa, or bar@news:woo, etc:

```
_sep_parts '(foo bar)' @ '(news ftp)' : '(woo laa)'
```

Complete some MAC addresses one octet at a time:

```
_multi_parts : '(00:11:22:33:44:55 00:23:34:45:56:67 00:23:45:56:67:78)'
```

Adding completion words directly using compadd

For more fine grained control you can use the builtin compadd function to add completion words directly. This function has many different options for controlling how completions are displayed and how text on the command line can be altered when words are completed. Read the [official documentation](#) for full details. Here I just give a few simple examples.

Add some words to the list of possible completions:

```
compadd foo bar blah
```

As above but also display an explanation:

```
compadd -X 'Some completions' foo bar blah
```

As above but automatically insert a prefix of “what_” before the completed word:

```
compadd -P what_ foo bar blah
```

As above but automatically insert a suffix of “_todo” after the completed word:

```
compadd -S _todo foo bar blah
```

As above but automatically remove the “_todo” suffix if a blank char is typed after the suffix:

```
compadd -P _todo -q foo bar blah
```

Add words in array \$wordsarray to the list of possible completions

```
compadd -a wordsarray
```

Testing & debugging

To reload a completion function:

```
> unfunction _func  
> autoload -U _func
```

The following functions can be called to obtain useful information. If the default keybindings don't work you can try pressing Alt+x and then enter the command name.

Function	Default keybinding	Description
_complete_help	Ctrl+x h	displays information about context names, tags, and completion functions used when completing at the current cursor position
_complete_help	Alt+2 Ctrl+x h	as above but displays even more information
_complete_debug	Ctrl+x ?	performs ordinary completion, but captures in a temporary file a trace of the shell commands executed by the completion system

Gotchas (things to watch out for)

Remember to include a #compdef line at the beginning of the file containing the completion function.

Take care to use the correct type of quoting for specifications to `_arguments` or `_regex_arguments`: use double quotes if there is a parameter that needs to be expanded in the specification, single quotes otherwise, and make sure to use different quotes around item descriptions.

Check that you have the correct number of `:`'s in the correct places for specifications for `_arguments`, `_alternative`, `_regex_arguments`, etc.

Remember to include an initial pattern to match the command word when using `_regex_arguments` (it does not need a matching action).

Remember to put a null char `$'\0'` at the end of any PATTERN argument for `_regex_arguments`

Tips

Sometimes you have a situation where there is just one option that can come after a subcommand, and zsh will complete this automatically when tab is pressed after the subcommand. If instead you want it listed with its description before completing you can add another empty option (i.e. `\:`) to the ACTION like this `' :TAG:DESCRIPTION: ((opt1\:"description for opt1" \:))'` Note this only applies to utility functions that use ACTIONS in their specification arguments (`_arguments`, `_regex_arguments`, etc.)

Other resources

[Here](#) is a nicely formatted short tutorial showing basic usage of the `_arguments` function, and [here](#) is a slightly more advanced tutorial using the `_arguments` function. [Here](#) is the `zshcompsys` man page.

[Give feedback](#)