Chapter 6: Completion, old and new

Completion of command arguments is something zsh is particularly good at. The simplest case is that you hit `<TAB>`, and the shell guesses what has to go there and fills it in for you:

```
  % ls
  myfile  theirfile  yourfile
  % cat t<TAB>
```

expands the command line to

```
  % cat theirfile
```

and you only had to type the initial letter, then `TAB`.

In the early days when this feature appeared in the C shell, only filenames could be completed; there were no clever tricks to help you if the name was ambiguous, it simply printed the unambiguous part and beeped so that you had to decide what to do next. You could also list possible completions; for some reason this became attached to the `^D` key in csh, which in later shells with Emacs-like bindings also deletes the next character, so that history has endowed zsh, like other shells, with the slightly odd combined behaviour:

```
  % cat yx
```

Now move the cursor back one character onto the x and hit ^D twice and you see: `yourfile`. That doesn't work if you use vi-like bindings, or, obviously, if you've rebound `^D`.

Next, it became possible to complete other items such as names of users, commands or hosts. Then zsh weighed in with menu completion, so you could keep on blindly hitting `<TAB>` until the right answer appeared, and never had to type an extra character yourself.

The next development was tcsh's, and then zsh's, programmable completion system; you could give instructions to the shell that in certain contexts, only certain items should be completed; for example, after `cd`, you would only want directories. In tcsh, there was a command called `complete`; each `` `complete ...' `` statement defined the completion for the arguments of a particular command, such as `cd`; the equivalent in zsh is `compctl`, which was inspired by `complete` but is different in virtually every significant detail. There is a perl script `lete2ctl` in the `Misc` directory of the shell distribution to help you convert from the tcsh to the zsh formats. You put a whole series of `compctl` commands into `.zshrc`, and everything else is done by the shell.

Zsh's system has become more and more sophisticated, and in version 3.1.6 a new completion system appeared which is supposed to do everything for you: you simply call a function, `compinit`, from an initialization file, after which zsh knows, for example, that `gunzip` should be followed by files ending in `.gz`. The new system is based on shell functions, an added bonus since they are extremely flexible and you already know the syntax. However, given the complexity it's quite difficult to get started writing your own completions now, and hard enough to know what to do to change the settings the way you like. The rest of the chapter should help.

I shall concentrate on the new completion system, which seems destined to take over completely from the old one eventually, now that the 3.1 release series has become the 4.0 production release. The old **compctl** command is still available, and old completion definitions will remain working in future versions of zsh --- in fact, on most operating systems which support dynamically linked libraries the old completion system is in a different file, which the shell loads when necessary, so there's very little overhead for this.

The big difference in the new system is that, instead of everything being set up once and for all when the shell starts, various bits of shell code are called after you hit `<TAB>`, to generate the completions there and then. There's enough new in the shell that all those unmemorable options to `compctl` (``-f`' for files ``-v`' for variables and so on) can be replaced by commands that produce the list of completions directly; the key command in this case is called ``compadd`', which is passed this list and decides what to use to complete the word on the command line. So the simplest possible form of new completion looks roughly like this:

```
  # tell the shell that the function mycompletion can do completion
  # when called by the widget name my-completion-widget, and that
  # it behaves like the existing widget complete-word
  zle -C my-completion-widget .complete-word mycompletion

  # define a key that calls the completion widget
  bindkey '^x^i' my-completion-widget

  # define the function that will be called
  mycompletion() {
    # add a list of completions
    compadd alpha bravo charlie delta
  }
```

That's very roughly what the completion system is doing, except that the function is called `_main_complete` and calls a lot of other functions to do its dirty work based on the context where completion was called (all the things that `compctl` used to do), and the widgets are just the old completion widgets (``expand-or-complete`' etc.) redefined and still bound to all the original keys. But, in case you hadn't guessed, there's more to it than that.

Here's a plan for the sections of this chapter.

1. A broad description of completion and expansion, applying equally to old and new completion.
2. How to configure completion using shell options. Most of this section applies to old completion, too, although I won't explicitly flag up any differences. After this, I shall leave the `compctl` world behind.
3. How to start up new completion.
4. The basics of how the new completion system works.
5. How to configure it using the new ``zstyle`' builtin.
6. Separate commands which do something other than the usual completion system, as well as some other editing widgets that have to do with completion.
7. Matching control, a powerful way of deciding such things as whether to complete case-insensitively, to allow insertion of extra parts of words before punctuation characters, or to ignore certain characters in the word on the command line.
8. How to write your own completion functions; you won't need to have too solid an understanding of all the foregoing just to do simple completions, but I will gradually introduce the full baroque splendour of how to make tags and styles work in your own functions, and how to make completion do the work of handling command arguments and options.
9. Ends the chapter gracefully, on the old `beginning, middle, end' principle.

6.1: Completion and expansion

More things than just completion happen when you hit tab. The first thing that zsh tries to do is expand the line. Expansion was covered in a previous chapter: basically all the things described there are possible candidates for expanding in-line by the editor. In other words, history substitutions with bangs, the various expansions using ``$`' or backquote, and filename generation (globbing) can all take place, with the result replacing what was there on the command line:

```
  % echo $PWD<TAB>
    ->    echo /home/pws/zsh/projects/zshguide
  % echo `print $ZSH_VERSION`<TAB>
    ->    echo 3.1.7
  % echo !!<TAB>
    ->    echo echo 3.1.7
  % echo ~/.z*<TAB>
    ->    echo /home/pws/.zcompdump /home/pws/.zlogout
          /home/pws/.zshenv /home/pws/.zshrc
```

Note that the `~' also gets expanded in this case.

This is often a good time to remember the `undo' key, `^_' or `^Xu'; typing this will restore what was there before the expansion if you don't like the result. Many keyboards have a quirk that what's described as `^_' should be typed as control with slash, which you'd write `^/' except unfortunately that does something else; this is not zsh's fault. There's another half-exception, namely filename generation: paths like `~/file' don't get expanded, because you usually know what they refer to and it's usually convenient to leave them for use in completion. However, the `=cmdname' form does get expanded, unless you have NO_EQUALS set.

In fact, deciding whether expansion or completion takes place can sometimes be tricky, since things that would be expanded if they were complete, may need to be completed first; for example $PAT should probably be completed to $PATH, but it's quite possible there is a parameter $PAT too. You can decide which, if you prefer. First, the commands expand-word, bound to `^X*', and the corresponding command for listing what would be expanded, list-expand, bound to `^Xg', do expansion only --- all possible forms except alias expansion, including turning `~/file' into a full path.

From the other point of view, you can use commands other than expand-or-complete, the one bound by default to <TAB>, to perform only completion. The basic command for this is complete-word, which is not bound by default. It is quite sensible to bind this to `^I' (i.e. <TAB>) if you are happy to use the separate commands for expansion, i.e.

```
  # Now tab does only completion, not expansion
  bindkey '^i' complete-word
```

Furthermore, if you do this and use the new completion system, then as we shall see there is a way of making the completion system perform expansion --- see the description of the _expand completer below. In this case you have much more control over what forms of expansion are tried, and at what point, but you have to make sure you use complete-word, not expand-or-complete, else the standard expansion system will take over.

There's a close relative of expand-or-complete, expand-or-complete-prefix, not bound by default. The only difference is that it will ignore everything under and to the right of the cursor when completing. It's as if there was a space where the cursor was, with everything to be ignored shifted to the right (guess how it's implemented). Use this if you habitually type new words in the line before other words, and expect them to complete or expand on their own even before you've typed the space after them. Some other shells work this way all the time. To be more explicit:

```
  % ls
  filename1
  % ls filex
```

Move the cursor to the x and hit tab. With expand-or-complete nothing happens; it's trying to complete a file called `filex' --- or, with the option COMPLETE_IN_WORD set, it's trying to find a file whose name starts with `file' and ends with `x'. If you do

```
  bindkey '^i' expand-or-complete-prefix
```

and try the same experiment, you will find the whole thing is completed to `filename1x', so that the `x' was ignored, but not removed.

One possible trap is that the listing commands, both `delete-char-or-list`, bound by default to `` `^D' `` in emacs mode, and `list-options`, bound by default to `` `^D' `` in vi insert mode and the basic command for listing completions as it doesn't have the delete-character behaviour, do not show possible expansions, so with the default bindings you can use `` `^D' `` to list, then hit `<TAB>` and find that the line has been completely rewritten by some expansion. Using `complete-word` instead of `expand-or-complete` will of course fix this. If you know how to write new editor widgets (chapter 4), you can make up a function which tries `list-expand`, and if that fails tries `list-options`.

There are four completion commands I haven't mentioned yet: three are `menu-complete`, `menu-expand-or-complete` and `reverse-menu-complete`, which perform menu completion, where you can cycle through all possible completions by hitting the same key. The first two correspond to `complete-word` and `expand-or-complete` respectively, while the third has no real equivalent as it takes you backwards through a completion list. The effect of the third can't be reached just by setting options for menu completion, so it's a useful one to bind separately. I have it bound to `` `\M-\C-i' ``, i.e. tab with the Meta key pressed down, but it's not bound by default.

The fourth is `menu-select`, which performs an enhanced form of menu completion called `menu selection' which I'll describe below when I talk about options. You have to make sure the `zsh/complist` module is loaded to use this zle command. If you use the style, zsh should be able to load this automatically when needed, as long as you have dynamic loading, which you probably do these days.

6.2: Configuring completion using shell options

There are two main ways of altering the behaviour of completion without writing or rewriting shell functions: shell options, as introduced in chapter 2, and styles, as introduced above. I shall first discuss the shell options, although as you will see some of these refer to the styles mechanism. Setting shell options affects every single completion, unless special care has been taken (using a corresponding style for the context, or setting an option locally) to avoid that.

In addition to the options which directly affect the completion system, completion is sensitive to various other options which describe shell behaviour. For example, if the option `MAGIC_EQUAL_SUBST` is set, so that arguments of all commands looking like `` `foo=~/file' `` have the `` `~' `` expanded as if it was at the start of an argument, then the default completion for arguments of commands not specially handled will try to complete filenames after the `=`.

Needless to say, if you write completion functions you will need to worry about a lot of other options which can affect shell syntax. The main starting point for completion chosen by context (everything except the commands for particular completions bound separately to keystrokes) is the function `_main_complete`, which includes the effect of the following lines to make sure that at least the basic options are set up within completion functions:

```
  setopt glob bareglobqual nullglob rcexpandparam extendedglob unset
  unsetopt markdirs globsubst shwordsplit shglob ksharrays cshnullglob
  unsetopt allexport aliases errexit octalzeroes
```

but that by no means exhausts the possibilities. Actually, it doesn't include those lines: the options to set are stored in the array `$_comp_options`, with `NO_` in front if they are to be turned off. You can modify this if you find you need to (and maybe tell the maintainers, too).

By the way, if you are wondering whether you can re-use the function `_main_complete`, by binding it to a different key with slightly different completion definitions, look instead at the description of the `_generic` command widget below. It's just a front-end to `_main_complete` which allows you to have a different set of styles in effect.

6.2.1: Ambiguous completions

The largest group of options deals with what happens when a completion is ambiguous, in other words there is more than one possible completion. The seven relevant options are as follows, as copied from the FAQ; many different combinations are possible:

- with `NO_BEEP` set, that annoying beep goes away,
- with `NO_LIST_BEEP`, beeping is only turned off for ambiguous completions,
- with `AUTO_LIST` set, when the completion is ambiguous you get a list without having to type `^D`,
- with `BASH_AUTO_LIST` set, the list only happens the second time you hit tab on an ambiguous completion,
- with `LIST_AMBIGUOUS`, this is modified so that nothing is listed if there is an unambiguous prefix or suffix to be inserted --- this can be combined with `BASH_AUTO_LIST`, so that where both are applicable you need to hit tab three times for a listing,
- with `REC_EXACT`, if the string on the command line exactly matches one of the possible completions, it is accepted, even if there is another completion (i.e. that string with something else added) that also matches,
- with `MENU_COMPLETE` set, one completion is always inserted completely, then when you hit TAB it changes to the next, and so on until you get back to where you started,
- with `AUTO_MENU`, you only get the menu behaviour when you hit TAB again on the ambiguous completion.

6.2.2: `ALWAYS_LAST_PROMPT`

The option `ALWAYS_LAST_PROMPT` is set by default, and has been since an earlier 3.1 release of zsh; after listing a completion, the cursor is taken back to the line it was on before, instead of reprinting it underneath. The downside of this is that the listing will be obscured when you execute the command or produce a different listing, so you may want to unset the option. `ALWAYS_LAST_PROMPT` behaviour is required for menu selection to work, which is why I mention it now instead of in the ragbag below.

When you're writing your own editor functions which invoke completion, you can actually cancel the effect of this with the widget `end-of-list`, which you would call as `zle end-of-list` (it's a normal editing function, not a completion function). You can also bind it to a key to use to preserve the existing completion list. On the other hand, if you want to control the behaviour within a completion function, i.e. to decide whether completion will try to return to the prompt above the list, you can manipulate it with the `last_prompt` element of the `$compstate` associative array, so for example:

```
  compstate[last_prompt]=''
```

will turn off the behaviour for the completion in progress. `$compstate` is the place to turn if you find yourself wanting to control completion behaviour in this much detail; see the `zshcompwid` manual page.

6.2.3: Menu completion and menu selection

The most significant matter decided by the options above is whether or not you are using menu completion. If you are not, you will need to type the next character explicitly when completion is ambiguous; if you are, you just need to keep hitting tab until the completion you want appears. In the second case, of course, this works best if there are not too many possibilities. Use of `AUTO_MENU` or binding the `menu-complete` widget to a separate key-stroke gives you something of both worlds.

A new variant of menu completion appeared in 3.1.6; in fact, it deserves the name menu completion rather more than the original form, but since that name was taken it is called `menu selection'. This allows you to move the cursor around the list of completions to select one. It is implemented by a separate module, `zsh/complist`; you can make sure this is loaded by putting `` `zmodload -i zsh/complist' `` in `.zshrc`, although it should be loaded automatically when the style `menu` is set as below. For it to be useful, you need two other things. The first is `ALWAYS_LAST_PROMPT` behaviour; this is suppressed if the whole completion list won't appear on the screen, since there's no line on the screen to go back to. However, menu selection does still work, by allowing you to scroll the list up and down. The second thing is that you need to start menu completion in any of the usual ways; menu selection is an addition to menu completion, not a replacement.

Now you should set the following style:

```
  zstyle ':completion:*' menu select=<NUM>
```

If an ambiguous completion produces at least `<NUM>` possibilities, menu selection is started. You can understand this best by trying it. One of the completions in the list, initially the top-leftmost, is highlighted and inserted into the line. By moving the cursor in the obvious directions (with wraparound at the edges), you change both the value highlighted and the value inserted into the line. When you have the value you want, hit return, which removes the list and leaves the inserted value. Hitting `^G` (the editor function `send-break`) aborts menu selection, removes the list and restores the command line.

Internally, zsh actually uses the parameter `$MENUSELECT` to supply the number and hence start menu selection. However, this is always initialised from the style as defined above, so you shouldn't set `$MENUSELECT` directly (unless you are using `compctl`, which will happily use menu selection). As with other styles, you can specify different values for different contexts; the `default` tag is checked if the current context does not produce a value for the style with whatever the current tag is. Note that the `menu` style also allows you to control whether menu completion is started at all, with or without selection; in other words, it is a style corresponding to the `MENU_COMPLETE` option.

There is one other additional feature when using menu selection. The zle command `accept-and-infer-next-history` has a different meaning here; it accepts a completion, and then tries to complete again using menu selection. This is very useful with directory hierarchies, and in combination with `undo` gives you a simple file browser. You need to bind it in the special keymap `menuselect`; for example, I use

```
  bindkey -M menuselect '^o' accept-and-infer-next-history
```

because the behaviour reminds me of what is usually bound to `^O` in emacs modes, namely `accept-line-and-down-history`. Binding it like this has no effect on `^O` in the normal keymaps. Try it out by entering menu selection on a set of files including directories, and typing `^O` on one of the directories. You should immediately have the contents of that directory presented for the next selection, while `undo` is smart enough not only to remove that selection but return to completion on the parent directory.

You can choose the manner in which the currently selected value in the completion list is highlighted using exactly the same mechanism as for specifying colours for particular types of matches; see the description of the `list-colors` style below.

6.2.4: Other ways of changing completion behaviour

`COMPLETE_ALIASES`

If you set an alias such as

```
  alias pu=pushd
```

then the alias `` `pu' `` will be expanded when the completion system is looking for the name of the command, so that it will instead find the command name `` `pushd' ``. This is quite useful to avoid having to define extra completions for all your aliases. However, it's possible you may want to define something different for the alias than for the command it expands to. In that case, you will need to set `COMPLETE_ALIASES`, and to make arrangements for completing after every alias which does not already match the name of a command. Hence `` `alias zcat="myzcat -dc"' `` will work with the option set, even if you haven't told the system about `` `myzcat' ``, while `` `alias myzcat="gzip -dc"' `` will not work unless you do define a completion for myzcat: here `` `compdef _gzip myzcat' `` would probably be good enough. Without the option set, it would be the other way around: the first alias would not work without the extra `compdef`, but the second would.

`AUTO_REMOVE_SLASH`

This option is turned on by default. If you complete a directory name and a slash is added --- which it usually is, both to tell you that you have completed a directory and to allow you to complete files inside it without adding a `` `/' `` by hand --- and the next thing you type is *not* something which would insert or complete part of a file in that directory, then the slash is removed. Hence:

```
 % rmdir my<TAB>
   ->  rmdir mydir/
 % rmdir mydir/<RETURN>
   ->  `rmdir mydir' executed
```

This example shows why this behaviour was added: some versions of `` `rmdir' `` baulk at having the slash after the directory name. On the other hand, if you continued typing after the slash, or hit tab again to complete inside `mydir`, then the slash would remain.

This is at worst harmless under most circumstances. However, you can unset the option `AUTO_REMOVE_SLASH` if you don't like that behaviour. One thing that may cause slight confusion, although it is the same as with other suffixes (i.e. bits which get added automatically but aren't part of the value being completed), is that the slash is added straight away if the value is being inserted by menu completion. This might cause you to think wrongly that the completion is finished, and hence is unique when in fact it isn't.

Note that some forms of completion have this type of behaviour built in, not necessarily with a slash, when completing lists of arguments. For example, enter `` `typeset ZSH_V<TAB>' `` and you will see `` `ZSH_VERSION=' `` appear, in case you want to assign something to the parameter; hitting space, which is not a possible value, makes the `` `=' `` disappear. This is not controlled by the `AUTO_REMOVE_SLASH` option, which applies only to directories inserted by the standard filename completion system.

`AUTO_PARAM_SLASH`, `AUTO_PARAM_KEYS`

These options come into effect when completing expressions with parameter substitutions. If `AUTO_PARAM_SLASH` is set, then any parameter expression whose value is the name of a directory will have a slash appended when completed, just as if the value itself had been inserted by the completion system.

The behaviour for `AUTO_PARAM_KEYS` is a bit more complicated. Try this:

```
  print ${ZSH_V<TAB>
```

You will find that you get the complete word `` `${ZSH_VERSION}' ``, with the closing brace and (assuming there are no other matching parameters) a space afterwards. However, often after you have completed a parameter in this fashion you want to type something immediately after it, such as a subscript. With `AUTO_PARAM_KEYS`, if you type something at this point which seems likely to have to go after the parameter name, it will immediately be put there without you having to delete the intervening characters --- try it with `` `[' ``, for example. Note that this only happens if the parameter name and any extra bits were added by completion; if you type everything by hand, typing `` `[' `` will not have this magic effect.

`COMPLETE_IN_WORD`

If this is set, completion always takes place at the cursor position in the word. For example if you typed `` `Mafile' ``, went back over the `` `f' ``, and hit tab, the shell would complete `` `Makefile' ``, instead of its usual behaviour of going to the end of the word and trying to find a completion there, i.e. something matching `` `Mafile*' ``. Some sorts of new completion (such as filename completion) seem to implement this behaviour regardless of the option setting; some other features (such as the `` `_prefix' `` completer described below) require it, so it's a good thing to set and get used to, unless you really need to complete only at the end of the word.

`ALWAYS_TO_END`

If this is set, the cursor is always moved to the end of the word after it is completed, even if completion took place in the middle. This also happens with menu completion.

6.2.5: Changing the way completions are displayed

`LIST_TYPES`

This is like the `-F` option to `ls`; files which appear in the completion listing have a trailing `` `/' `` for a directory, `` `*' `` for a regular file executable by the current process, `` `@' `` for a link, `` `|' `` for a named pipe, `` `%' `` for a character device and `` `#' `` for a block device. This option is on by default.

Note that the identifiers only appear if the completion system knows that the item is supposed to be a file. This is automatic if the usual filename completion commands are used. There is also an option `-f` to the builtin `compadd` if you write your own completion function and want to tell the shell that the values may be existing files to apply `LIST_TYPES` to (though no harm is caused if no such files exist).

`LIST_PACKED`, `LIST_ROWS_FIRST`

These affect the arrangement of the completion listing. With `LIST_PACKED`, completion lists are made as compact as possible by varying the widths of the columns, instead of formatting them into a completely regular grid. With `LIST_ROWS_FIRST`, the listing order is changed so that adjacent items appear along rows instead of down columns, rather like `ls`'s `-x` option.

It is possible to alter both these for particular contexts using the styles `list-packed` and `list-rows-first`. The styles in such cases always override the option; the option setting is used if no corresponding style is found.

Note also the discussion of completion groups later on: it is possible to have different types of completion appear in separate lists, which may then be formatted differently using these tag-sensitive styles.

6.3: Getting started with new completion

Before I go into any detail about new completion, here's how to set it up so that you can try it out. As I said above, the basic objects that do completions are shell functions. These are all autoloaded, so the shell needs to know where to find them via the `$fpath` array. If the shell was installed properly, and nothing in the initialization files has removed the required bits from `$fpath`, this should happen automatically. It's even possible your system sets up completion for you (Mandrake Linux 6.1 is the first system known to do this out of the box), in which case type `` `which compdef' `` and you should see a complete shell function --- actually the one which allows you to define additional completion functions. Then you can skip the next paragraph.

If you want to load completion, try this at the command line:

```
  autoload -U compinit
  compinit
```

which should work silently. If not, you need to ask your system administrator what has happened to the completion functions or find them yourself, and then add all the required directories to your `$fpath`. Either they will all be in one big directory, or in a set of subdirectories with the names `AIX`, `BSD`, `Base`, `Debian`, `Redhat`, `Unix`, `X` and `Zsh`; in the second case, all the directories need to be in `$fpath`. When this works, you can add the same lines, including any modification of `$fpath` you needed, to your `.zshrc`.

You can now see if it's actually working. Type `` `cd ' ``, then `^D`, and you should be presented with a list of directories only, no regular files. If you have `$cdpath` set, you may see directories that don't appear with `ls`. As this suggests, the completion system is supplied with completions for many common (and some quite abstruse) commands. Indeed, the idea is that for most users completion just works without intervention most of the time. If you think it should when it doesn't, it may be a bug or an oversight, and you should report it.

Another example on the theme of `it just works':

```
  tar xzf archive.tar.gz ^D
```

will look inside the gzipped tar archive --- assuming the GNU version of `tar`, for which the ``z`' in the first set of arguments reports that the archive has been compressed with gzip --- and give you a list of files or directories you can extract. This is done in a very similar way to normal file completion; although there are differences, you can do completion down to any directory depth within the archive. (At this point, you're supposed to be impressed.)

The completion system knows about more than just commands and their arguments, it also understands some of the shell syntax. For example, there's an associative array called `$_comps` which stores the names of commands as keys and the names of completion functions as the corresponding values. Try typing:

```
  print ${_comps[
```

and then `^D`. You'll probably get a message asking if you really want to see all the possible completions, i.e. the keys for `$_comps`; if you say ``y`' you'll see a list. If you insert any of those keys, then close the braces so you have e.g. ``${_comps[mozilla]}`' and hit return, you'll see the completion function which handles that command; in this case (at the time of writing) it's `_webbrowser`. This is one way of finding out what function is handling a particular command. If there is no entry --- i.e. the ``print ${_comps[mycmd]}`' gives you a blank line --- then the command is not handled specially and will simply use whatever function is defined for the ``-default-`' context, usually `_default`. Usually this will just try to complete file names. You can customize `_default`, if you like.

Apart from `-default-`, some other of those keys for `_comps` also look like `-this-`: they are special contexts, places other than the arguments of a command. We were using the context called `-subscript-`; you'll find that the function in this case is called `_subscript`. Many completion functions have names which are simply an underscore followed by the command or context name, minus any hyphens. If you want a taster of how a completion function looks, try ``which _subscript`'; you may well find there are a lot of other commands in there that you don't know yet.

It's important to remember that the function found in this way is at the root of how a completion is performed. No amount of fiddling with options or styles --- the stuff I'm going to be talking about for the next few sections --- will change that; if you want to change the basic completion, you will just have to write your own function.

By the way, you may have old-style completions you want to mix-in --- or maybe you specifically don't want to mix them in so that you can make sure everything is working with the new format. By default, the new completion system will first try to find a specific new-style completion, and if it can't it will try to find a `compctl`-defined completion for the command in question. If all that fails, it will try the usual new-style default completion, probably just filename completion. Note that specific new-style completions take precedence, which is fair enough, since if you've added them you almost certainly don't want to go back and use the old form. However, if you don't ever want to try old-style completion, you can put the following incantation in your `.zshrc`:

```
  zstyle ':completion:*' use-compctl false
```

For now, that's just black magic, but later I will explain the `style' mechanism in more detail and you will see that this fits in with the normal way of turning things off in new-style completion.

6.4: How the shell finds the right completions

6.4.1: Contexts

The examples above show that the completion system is highly context-sensitive, so it's important to know how these contexts are described. This system evolved gradually, but everything I say applies to all versions of zsh with the major version 4.

state we are at in completion, and is given as a sort of colon-separated path, starting with the least specific part. There's an easy way of finding out what context you are in: at the point where you want to complete something, instead type `` `^Xh' ``, and it will tell you. In the case of the `$_comps` example, you will find,

```
  :completion::complete:-subscript-::
```

plus a list of so-called `tags' and completion functions, which I'll talk about later. The full form is:

```
  :completion:<func>:<completer>:<command>:<argument>:<tag>
```

where the elements may be missing if they are not set, but the colons will always be there to make pattern matching easier. Here's what the bits of the context mean after the `:completion:` part, which is common to the whole completion system.

<func>
: is the name of a function from which completion is called --- this is blank if it was started from the standard completion system, and only appears in a few special cases, listed in section six of this chapter.

<completer>
: is called `` `complete' `` in this case: this refers to the fact that the completion system can do more than just simple completion; for example, it can do a more controlled form of expansion (as I mentioned), spelling correction, and completing words with spelling mistakes. I'll introduce the other completers later; `` `complete' `` is the simplest one, which just does basic completion.

<command>
: is the name of a command or other similar context as described above, here `` `-subscript-' ``.

<argument>
: is most useful when `<command>` is the name of a real command; it describes where in the arguments to that command we are. You'll see how it works in a moment. Many of the simpler completions don't use this; only the ones with complicated option and argument combinations. You just have to find out with `^Xh` if you need to know.

<tag>
: describes the type of a completion, essentially a way of discriminating between the different things which can be completed at the same point on the command line.

Now look at the context for a more normal command-argument completion, e.g. after `cd`; here you'll see the context `` `:completion::complete:cd::' ``. Here the command-name part of the context is a real command.

For something more complicated, try after `` `cvs add' `` (it doesn't matter for this if you don't have the `cvs` command). You'll see a long and repetitive list of tags, for two possible contexts,

```
  :completion::complete:cvs:argument-rest:
  :completion::complete:cvs-add:argument-rest:
```

The reason you have both is that the `` `add' `` is not only an argument to `cvs`, as the first context would suggest, it's also a subcommand in its own right, with its own arguments, and that's what the second context is for. The first context implies there might be more subcommands after `` `add' `` and its arguments which are completely separate from them --- though in fact CVS doesn't work that way, so that form won't give you any completions here.

In both, `` `argument-rest' `` shows that completion is looking for another argument, the `` `rest' `` indicating that it is the list of arguments at the end of the line; if position were important (see `` `cvs import' `` for an example), the context would contain `argument-1', or whatever. The `cvs-add' shows how subcommands are handled, by separating with a hyphen instead of a colon, so as not to confuse the different bits of the context.

Apart from arguments to commands and subcommands, arguments to options are another frequent possibility; for an example of this, try typing ^Xh after `dvips -o' and you will see the context `:completion::complete:dvips:option-o-1:'; this shows you are completing the first argument to dvips's -o option, (it only takes one argument) which happens to be the name of a file for output.

6.4.2: Tags

Now on to the other matter to do with contexts, tags. Let's go back and look at the output from the ^Xh help test after the cd command in full:

```
  tags in context :completion::complete:cd::
    local-directories path-directories  (_alternative _cd)
```

Unlike the contexts considered so far, which tell you how completion arrived at the point it did, the tags describe the things it can complete here. In this case, there are three: directory-stack refers to entries such as `+1'; the directory stack is the set of directories defined by using the pushd command, which you can see by using the dirs command. Next, local-directories refers to subdirectories of the current working directory, while path-directories refers to any directories found by searching the $cdpath array. Each of the possible completions which the system offers belongs to one of those classes.

In parentheses, you see the names of the functions which were called to generate the completions; these are what you need to change or replace if you want to alter the basic completion behaviour. Calling functions appear on the right and called functions on the left, so that in this case the function `_cd' was the function first called to handle arguments for the cd command, fitting the usual convention. Some standard completion functions have been filtered out of this list --- it wouldn't help you to know it had been through _main_complete and _complete, for example.

Maybe it's already obvious that having the system treat different types of completion in different ways is useful, but here's an example, which gives you a preview of the `styles' mechanism, discussed later. Styles are a sort of glorified shell parameter; they are defined with the zstyle command, using a style name and possible values which may be an array; you can always define a style as an array, but some styles may simply use it as a string, joining together the arguments you gave it with spaces. You can also use the zstyle command, with different arguments, to retrieve their value, which is what the completion system itself does; there's no actual overlap with parameters and their values, so they don't get in the way of normal shell programming.

Where styles differ from parameters is that they can take different values in different contexts. The first argument to the zstyle command gives a context; when you define a style, this argument is actually a pattern which will be matched against the current context to see if the style applies. The rule for finding out what applies is: exact string matches are preferred before patterns, and longer patterns are preferred before shorter patterns. Here's that example:

```
  zstyle ':completion:*:cd:*' tag-order local-directories \
    path-directories
```

From the discussion of contexts above, the pattern will match any time an argument to the cd command is being completed. The style being set is called tag-order, and the values are the two tags valid for directories in cd.

The tag-order style determines the order in which tags are tried. The value given above means that first local-directories will be completed; only if none can be completed will path-directories be tried. You can enter the command and try this; if you don't have $cdpath set up you can assign `cdpath=(~)', which will allow `cd foo' to change to a directory `~/foo' and allow completion of directories accordingly. Go to a directory other than ~; completion for cd will only show subdirectories of where you are, not those of ~, unless you type a string which is the prefix of a directory under ~ but not your current directory. For example,

```
% cdpath=(~)
% ls -F ~
foo/    bar/
% ls -F
rod/    stick/
# Without that tag-order zstyle command, you would get...
% cd ^D
bar/    foo/    rod/    stick/
% zstyle ':completion:*:cd:*' tag-order local-directories \
   path-directories
# now you just get the local directories, if there are any...
% cd ^D
rod/    stick/
```

There's more you can do with the `tag-order` style: if you put the tags into the same word by quoting, for example `"local-directories path-directories"`, then they would be tried at the same time, which in this case gives you the effect of the default. In fact, since it's too much work to know what tags are going to be available for every single possible completion, the default when there is no appropriate `tag-order` is simply to try all the tags available in the context at once; this was of course what was originally happening for completion after `cd`.

Even if there is a `tag-order` specification, any tags not specified will usually be tried all together at the end, so you could actually have missed out `path-directories` from the end of the original example and the effect would have been the same. If you don't want that to happen, you can specify a `` `-' `` somewhere in the list of tags, which is not used as a tag but tells completion that only the tags in the list should be tried, not any others that may be available. Also, if you don't want a particular tag to be shown you can include `` `!tagname' `` in the values, and all the others but this will be included. For example, you may have noticed that when completing in command position you are offered parameters to set as well as commands etc.:

```
Completing external command
tex             texhash         texi2pdf        text2sf
texconfig       texi2dvi        texindex        textmode
texdoc          texi2dvi4a2ps   texlinks        texutil
texexec         texi2html       texshow         texview
Completing parameter
TEXINPUTS                               texinputs
```

(I haven't told you how to produce those descriptions, or how to make the completions for different tags appear separately, but I will --- see the descriptions of the `` `format' `` and `` `group-name' `` styles below.) If you set

```
zstyle ':completion:*:-command-:*' tag-order '!parameters'
```

then the last two lines will disappear from the completion. Of course, your completion list probably looks completely different from mine anyway. By the way, one good thing about styles is that it doesn't matter whether they're defined before or after completion is loaded, since styles are stored and retrieved by another part of the shell.

To exclude more than one tag name, you need to include the names in the same word. For example, to exclude both parameters and reserved words the value would be `'!parameters reserved-words'`, and *not* `'!parameters' '!reserved-words'`, which would try completion once with parameters excluded, then again with reserved words excluded. Furthermore, tags can actually be patterns, or more precisely any word in one of the arguments to `tag-order` may contain a pattern, which will then be tried against all the valid tags to see if it matches. It's sometimes even useful to use `*' to match all tags, if you are specifying a special form of one of the tags --- maybe using a label, as described next --- in the same word. See the manual for all the tag names understood by the supplied functions.

The `tag-order` style allows you to give tags `labels', which are a sort of alias, instructing the completion system to use a tag under a different name. You arrange this by giving the tag followed by a colon, followed by the label. The label can also have a hyphen in front, which means that the original tag name should be put in front when the label is looked up; this is really just a way of making the names look neater. The upshot is that by using contexts with the label name in, rather than the tag name, you can arrange for special behaviour. Furthermore, you can give an alternative description for the labelled tag; these show up with the `format` style which I'll describe below (and which I personally find very useful). You put the description after another colon, with any spaces quoted. It would look like this:

```
  zstyle ':completion:*:aliens:*' tag-order \
  'frooble:-funny:funny\ frooble' frooble
```

which is used when you're completing for the command `aliens`, which presumably has completions tagged as \``frooble`' (if not, you're very weird). Then completion will first look up styles for that tag under the name `frooble-funny`, and if it finds completions using those styles it will list them with a description (if you are using `format`) of \`funny frooble'. Otherwise, it will look up the styles for the tag under its usual name and try completion again. It's presumably obvious that if you don't have different styles for the two labels of the tag, you get the same completions each time.

Rather than overload you with information on tags by giving examples of how to use tag labels now, I'll reserve this for the description of the `ignored-patterns` style below, which is one neat use for labels. In fact, it's the one for which it was invented; there are probably lots of other ones we haven't thought of yet.

One important note about `tag-order` which I may not have made as explicit as I should have: *it doesn't change which tags are actually valid in that completion*. Just putting a tag name into the list doesn't mean that tag name will be used; that's determined entirely by the completion functions for a particular context. The `tag-order` style simply alters the order in which the tags which *are* valid are examined. Come back and read this paragraph again when you can't work out why `tag-order` isn't doing what you want.

Note that the rule for testing patterns means that you can always specify a catch-all worst case by \``zstyle "*" style ...`', which will always be tried last --- not just in completion, in fact, since other parts of the shell use the styles mechanism, and without the \``:completion:`' at the start of the context this style definition will be picked up there, too.

Styles like `tag-order` are the most important case where tags are used on their own. In other cases, they can be added to the end of the context; this is useful for styles which can give different results for different sets of completions, in particular styles that determine how the list of completions is displayed, or how a completion is to be inserted into the command line. The tag is the final element, so is not followed by a colon. A full context then looks something like \``:completion::complete:cd::path-directories`'. Later, you'll see some styles which can usefully be different for different tag contexts. Remember, however, that the tags part of the context, like other parts, may be empty if the completion system hasn't figured out what it should be yet.

6.5: Configuring completion using styles

You now know how to define a style for a particular context, using

```
  zstyle <context> <style> <value...>
```

and some of the cases where it's useful. Before introducing other styles, here's some more detailed information. I already said that styles could take an array value, i.e. a set of values at the end of the `zstyle` command corresponding to the array elements, and you've already seen one case (`tag-order`) where that is useful. Many styles only use one value, however. There is a particularly common case, where you simply want to turn a value on or off, i.e. a boolean value. In this case, you can use any of \``true`', \``yes`', \``on`' or \``1`' for on and \``false`', \``no`', \``off`' or \``0`' for off. You define all styles the same way; only when they're used is it decided whether they should be a scalar, an array, or a boolean, nor is the name of a style checked to see if it is valid, since the shell doesn't know what styles might later be looked up. The same obviously goes for contexts.

You can list existing styles (not individually, only as a complete list) using either `zstyle' or `zstyle -L'. In the second case, they are output as the set of zstyle commands which would regenerate the styles currently defined. This is also useful with grep, since you can easily check all possible contexts for a particular style.

The most powerful way of using zstyle is with the option -e. This says that the words you supply are to be evaluated as if as arguments to eval. This should set the array $reply to the words to be used. So

```
  zstyle '*' days Monday Tuesday
```

and

```
  zstyle -e '*' days 'reply=(Monday Tuesday)'
```

are equivalent --- but the intention, of course, is that in the second case the argument can return a different value each time so that the style can vary. It will usually be evaluated in the heat of completion, hence picking up all the editing parameters; so for example

```
  zstyle -e ':completion:*' mystyles 'reply=(${NUMERIC:-0})'
```

will make the style return a non-zero integer (possibly indicating true) if you entered a non-zero prefix argument to the command, as described in chapter 4. However, the argument can contain any zsh code whatsoever, not just a simple assignment. Remember to quote it to prevent it from being turned into something else when the zstyle command line is run.

Finally, you can delete a context for a style or a list of styles by

```
  zstyle -d [ <context-pattern> [ <style> ] ] ...
```

--- note that although the first argument is a pattern, in this case it is treated exactly, so if you give the pattern `:completion:*:cd:*', only values given with *exactly* that pattern will be deleted, not other values whose context begins with `:completion:' and contains `:cd:'. The pattern and the style are optional when deleting; if omitted, all styles for the context, or all styles of any sort, are deleted. The completion system has its own defaults, but these are builtin, so anything you specify takes precedence.

By the way, I did mention in passing in chapter 4 that you could use styles in just the same way in ordinary zle widgets (the ones created with `zle -N'), but you probably forgot about that straight away. All the instructions about defining styles and using them in your own functions from this chapter apply to zle functions. The only difference is that in that case the convention for contexts is that the context is set to `:zle:*widget-name*' for executing the widget *widget-name*.

The rest of this section describes some useful styles. It's up to you to experiment with contexts if you want the style's values to be different in different places, or just use `*' if you don't care.

6.5.1: Specifying completers and their options

`Completers' are the behind-the-scenes functions that decide what sort of completion is being done. You set what completers to use with the `completer' style, which takes an array of completers to try in order. For example,

```
  zstyle ':completion:*' completer _complete _correct _approximate
```

specifies that first normal completion will be tried (`_complete'), then spelling correction (`_correct'), and finally approximate completion (`_approximate'), which is essentially the combined effect of the previous two, i.e. complete the word typed but allow for spelling mistakes. All completers set the context, so inside _complete you will usually find `:completion::complete:...', inside correction `:completion::correct:..', and so on.

There's a labelling feature for completers, rather like the one for tags described, but not illustrated in detail, above. You can put a completer in a list like this:

```
  zstyle ':completion:*' completer ... _complete:comp-label ...
```

which calls the completer `_complete`, but pretends its name is `comp-label` when looking things up in styles, so you can try completers more than once with different features enabled. As with tags, you can write it like `` `_complete:-label' ``, and the normal name will be prepended to get the name `` `complete-label' `` --- just a shortcut, it doesn't introduce anything new. I'll defer an example until you know what the completers do.

Here is a more detailed description of the existing completers; they are all functions, so you can simply copy and modify one to make your own completer.

`_complete`

This is the basic completion behaviour, which we've been assuming up to now. Its major use is simply to check the context --- here meaning whether we are completing a normal command argument or one of the special `` `-context-' `` places --- and call the appropriate completion function. It's possible to trick it by setting the parameter `` `compcontext' `` which will be used instead of the one generated automatically; this can be useful if you write your own completion commands for special cases. If you do this, you should make the parameter local to your function.

`_approximate`

This does approximate completion: it's actually written as a wrapper for the `_complete` completer, so it does all the things that does, but it also sets up the system to allow completions with misspellings. Typically, you would want to try to complete without misspellings first, so this completer usually appears after `_complete` in the `completers` style.

The main means of control is via the `max-errors` style. You can set this to the maximum number of errors to allow. An error is defined as described in the manual for approximate pattern matching: a character missing such as `` `rhythm' `` / `` `rhytm' ``, an extra character such as `` `rhythm' `` / `` `rhythms' ``, an incorrect character such as `` `rhythm' `` / `` `rhxthm' ``, or a pair of characters transposed such as `` `rhythm' `` `` `rhyhtm' `` each count as one error. Approximation will first try to find a match or matches with one error, then two errors, and so on, up to and including the value of `max-errors`; the set of matches with the lowest number of errors is chosen, so that even if you set `max-errors` large, matches with a lower number of errors will always be preferred. The real problems with setting a large `max-errors` are that it will be slower, and is more likely to generate matches completely unlike what you want --- with typing errors, two or three are probably the most you need. Otherwise, there's always Mavis Beacon. Hence:

```
  % zstyle ':completion:*' max-errors 2
  # just for the sake of example...
  % zstyle ':completion:*' completer _approximate
  % ls
  ashes    sackcloth
  % echo siccl<TAB>
    -> echo sackcloth
  % echo zicc<TAB>
    <Beep.>
```

because `` `s[i/a]c[k]cloth' `` is only two errors, while `` `[z/s][i/a]c[k]cloth' `` would be three, so doesn't complete.

There's another way to give a maximum number of errors, using the numeric prefix specified with `ESC-<digit>` in Emacs mode, directly with number keys in vi command mode, or with `universal-argument`. To enable this, you have to include the string `numeric` as one of the values for `max-errors` --- hence this can actually be an array, e.g.

```
  zstyle ':completion:*:approximate:*' max-errors 2 numeric
```

allows up to two errors automatically, but you can specify a higher maximum by giving a prefix to the completion command. So to continue the example above, enter the new `zstyle` and:

```
  % echo zicc<ESC-3><TAB>
    -> echo sackcloth
```

because we've allowed three errors. You can start to see the problems with allowing too many errors: if you had the file `` `zucchini' ``, that would be only one error away, and would be found and inserted before `` `sackcloth' `` was even considered.

Note that the context is examined straightaway in the completer, so at this stage it is simply `` `:completion::approximate:::' ``; no more detailed contextual information is available, so it is not possible to specify different `max-errors` for different commands or tags.

The final possibility as a value for the style is `` `not-numeric' ``: that means if any numeric prefix is given, approximation will not be done at all. In the last example, completion would have to find a file beginning `` `zicc' ``.

Other minor styles also control approximation. The style `original`, if true means the original value is always treated as a possible completion, even if it doesn't match anything and even if nothing else matched. Completing the original and the corrections use different tags, unimaginatively called `original` and `corrections`, so you can organise this with the `tag-order` style.

Because the completions in this case usually don't match what's already on the command line, and may well not match each other, menu completion is entered straight away for you to pick a completion. You can arrange that this doesn't happen if there is an unambiguous piece at the start to insert first by setting the boolean style `insert-unambiguous`.

Those last two styles (`original` and `insert-unambiguous`) are looked up quite early on, when the context for generating corrections is being set up, so that only the context up to the completer name is available. The completer name will be followed by a hyphen and the number of errors currently being accepted. So for trying approximation with one error the context is `` `:completion::approximate-1:::' ``; if that fails and the system needs to look for completion with two errors, the context will be `` `:completion::approximate-2:::' ``, and so on; the same happens with correction and `` `correct-1' ``, etc., for the completer described next.

`_correct`

This is very similar to `_approximate`, except that the context is `` `:completion::correct:*' `` (or `` `:completion::correct-<num>:*' `` when generating corrections, as described immediately above) and it won't perform completion, just spelling correction, so extra characters which the completer has to add at the end of the word on the line now count as extra errors instead of completing in the ordinary way: `zicc` is woefully far from `sackcloth`, seven errors, but `ziccloth` only counts three again. The `_correct` completer is controlled in just the same way as `_approximate`.

There is a separate command which only does correction and nothing else, usually bound to `` `^Xc' ``, so if you are happy using that you don't need to include `_correct` in the list of completers. If you do include it, and you also have `_approximate`, `_correct` should come earlier; `_approximate` is bound to generate all the matches `_correct` does, and probably more. Like other separate completion commands, it has its own context, here beginning `` `:completion:correct-word:' ``, so it's easy to make this command behave differently from the normal completers.

Old-timers will remember that there is another form of spelling correction built into the shell, called with `` `ESC-$' `` or `` `ESC-s' ``. This only corrects filenames and doesn't understand anything about the new completion mechanism; the only reason for using it is that it may well be faster. However, if you use the `CORRECT` or `CORRECT_ALL` shell options, you will be using the old filename correction mechanism; it's not yet possible to alter this.

`_expand`

This actually performs expansion, not completion; the difference was explained at the start of the chapter. If you use it, you should bind tab to `complete-word`, not `expand-or-complete`, since otherwise expansion will be performed before the completion mechanism is started up. As expansion should still usually be attempted before completion, this completer should appear before `_complete` and its relatives in the list of values for the `completers` style.

The reason for using this completer instead of normal expansion is that you can control which expansions are performed using styles in the \``:completion:*:expand:*`' context. Here are the relevant styles:

`glob`
: expands glob expressions, in other words does filename generation using wildcards.

`substitute`
: expands expressions including and active \``$`' or backquotes.

But remember that you need

```
  bindkey '^i' complete-word
```

when using this completer as otherwise the built-in expansion mechanism which is run by the normal binding `expand-or-complete` will take over.

You can also control how expansions are inserted. The tags for adding expansions are `original` (presumably self-explanatory), `all-expansions`, which refers to adding a single string containing all the possible expansions (the default, just like the editor function `expand-word`), and `expansions`, which refers to the results added one by one. By changing the order in which the tags are tried, as described for the `tag-order` style above, you can decide how this happens. For example,

```
  zstyle ':completion:*' completer _expand _complete
  zstyle ':completion::expand:*' tag-order expansions
```

sets up for performing glob expansion via completion, with the expansions being presented one by one (usually via menu completion, since there is no common prefix). Altering `expansions` to `all-expansions` would insert the list, as done by the normal expansion mechanism, while altering it to \``expansions original`' would keep the one-at-a-time entry but also present the original string as a possibility. You can even have all three, i.e. the entire list as a single string becomes just one of the set of possibilities.

There is also a `sort` style, which determines whether the expansions generated will be sorted in the way completions usually are, or left just as the shell produced them from the expansion (for example, expansion of an array parameter would produce the elements in order). If it is `true`, they will always be sorted, if `false` or unset never, and if it is `menu` they will be sorted for the `expansions` tag, but not for the `all-expansions` tag which will be a single string of the values in the original order.

There is a slight problem when you try just to generate `glob` expansions, without `substitute`. In fact, it doesn't take much thought to see that an expression like \``$PWD/*.c`' doesn't mean anything if `substitute` is inactive; it must be active to make sense of such expressions. However, this is annoying if there are no matches: you end up being offered a completion with the expanded `$PWD`, but \``*.c`' still tacked on the end, which isn't what you want. If you use `_expand` mainly for globbing, you might therefore want to set the style `subst-globs-only` to true: if a completion just expands the parameters, and globbing does nothing, then the expansion is rejected and the line left untouched.

The `_expand` completer will also use the styles

`accept-exact`
: applies to words beginning with a \``$`' or \``~`'. Suppose there is a parameter \``$foo`' and a parameter \``$foobar`' and you have \``$foo`' on the line. Normally the completion system will perform completion at this point. However, with `accept-exact` set, \``$foo`' will be expanded since it matches a parameter.

add-space

means add a space after the expansion, as with a successful completion --- although directories are given a \``/`' instead. For finer control, it can be set to the word `file`, which means the space is only added if the expanded word matches a file that already exists (the idea being that, if it doesn't, you may want to complete further). Both `true` and `file` may be combined with `subst`, which prevents the adding of a space after expanding a substitution of the form \``${...}`' or \``$(...)`'.

keep_prefix

also addresses the question of whether a \``~`' or \``$`' should be expanded. If set, the prefix will be retained, so expanding \``~/f*`' to \``~/foo`' doesn't turn the \``~`' into \``/home/pws`'. The default is the value \``changed`', which is a half-way house been `false` and `true`: it means that if there was no other change in the word, i.e. no other possible expansion was found, the \``~`' or \``$`' will be expanded. If the effect of this style is that the expansion is the same as the unexpanded word, the next completer in the list after `_expand` will be tried.

suffix

is similar to `keep_prefix`. The \`suffix' referred to is something after an expression beginning \``~`' or \``$`' that wouldn't be part of that expansion. If this style is set, and such a suffix exists, the expansion is not performed. So, for example, \``~pw<TAB>`' can be expanded to \``~pws`', but \``~pw/`' is not eligible for expansion; likewise \``$fo`' and \``$fo/`'. This style defaults to `true` --- so if you want `_expand` always to expand such expressions, you will need to set it to `false` yourself.

An easier way of getting the sort of control over expansion which the `_expand` completer provides is with the `_expand_word` function, usually bound to `\C-xe`, which does all the things described above without getting mixed up with the other completers. In this case the context string starts \``:completion:expand-word`', so you can have different styles for this than for the `_expand` completer.

Setting different priorities for expansion is one good use for completer labels, for example

```
  zstyle ':completion:*' completer _expand:-glob _expand:-subst
  zstyle ':completion:*:expand-glob:*' glob yes
  zstyle ':completion:*:expand-subst:*' substitute yes
```

is the basic set up to make `_expand` try glob completions and failing that do substitutions, presenting the results as an expansion. You would almost certainly want to add details to help this along.

_history

This completes words from the shell's history, in other words everything you typed or had completed or expanded on previous lines. There are three styles that affect it, `sort` and `remove-all-dups`; they are described for the command widget `_history_complete_word` below. That widget essentially performs the work of this completer as a special keystroke.

_prefix

Strictly, this completer doesn't do completion itself, and should hence be in the group below starting with `_match`. However, it *seems* to do completion... Let me explain.

Many shells including zsh have the facility to complete only the word before the cursor, which zsh completion jargon refers to as the \`prefix'. I explained this above when I talked about `expand-or-complete-prefix`; when you use that instead of the normal completion functions, the word as it's finally completed looks like \``<prefix><completion><suffix>`' where the completion has changed \``<prefix>`' to \``<prefix><completion>`', ignoring `<suffix>` throughout.

The `_prefix` completer lets you do this as part of normal completion. What happens is that the completers are evaluated as normal, from left to right, until a completion is found. If `_prefix` is reached, completion is then attempted just on the prefix. So if your completers are \``_complete _prefix`', the shell will first try completion on the whole word, prefix and suffix, then just on the prefix. Only the first `real' completer (_complete, _approximate, _correct, _expand, _history) is used.

You can try prefix completion more than once simply by including _prefix more than once in the list of completers; the second time, it will try the second `real' completer in the list; so if they are `_complete _prefix _correct _prefix', you will get first ordinary completion, then the same for the prefix only, then ordinary correction, then the same for the prefix only. You can move either of the _prefix completers to the point in the sequence where you want the prefix-only version to be tried.

The _prefix completer will re-look up the completer style. This means that you can use a non-default set of completers for use just with _prefix. Here, as described in the manual, is how to force _prefix only to be used as a last resort, and only with normal completion:

```
  zstyle ':completion:::::' completer _complete \
    <other-completers> _prefix
  zstyle ':completion::prefix:::' completer _complete
```

The full contexts are shown, just to emphasise the form; as always, you can use wildcards if you don't care. In a case like this, you can use *only* _prefix as the completer, and completion including the suffix would never be tried; you then have to make sure you have the completer style for the prefix context, however, or no completion at all will be done.

The completer labelling trick is again useful here: you can call _prefix more than once, wherever you choose in your list of completers, and force it to look up in a different context each time.

```
  zstyle ':completion:*' completer _complete _prefix:-complete \
    _approximate _prefix:-approximate
  zstyle ':completion:*:prefix-complete:*' completer _complete
  zstyle ':completion:*:prefix-approximate:*' completer _approximate
```

This tries ordinary completion, then the same for the prefix only, then approximation, then the same for the prefix only. As mentioned in the previous paragraph, it is perfectly legitimate to leave out the raw _complete and _approximate completers and just use the forms with the _prefix prefix.

One gotcha with the _prefix completer: you have to make sure the option COMPLETE_IN_WORD is set. That may sound counter-intuitive: after all, _prefix forces completion *not* to complete inside a word. The point is that without that option, completion is only ever tried at the end of the word, so when you type <TAB> in the middle of <prefix><suffix>, the cursor is moved to after the end of the suffix before the completion system has a chance to see what's there, and hence the whole thing is regarded as a prefix, with no suffix.

There's one more style used with _prefix: `add-space'. This makes _prefix add a real, live space when it completes the prefix, instead of just pretending there was one there, hence separating the completed word from the original suffix; otherwise it would simply leave the resulting word all joined together, as expand-or-complete-prefix usually does.

_ignored

Like _prefix this is a bit of a hybrid, mopping up after completions which have already been generated. It allows you to have completions which have already been rejected by the style `ignored-patterns'. I'll describe that below, but it's effect is very simple: for the context given, the list of patterns you specify are matched against possible completions, and any that match are removed from the list. The _ignored completer allows you to retrieve those removed completions later in your completer list, in case nothing else matched.

This is used by the $fignore mechanism --- a list of suffixes of files not normally to be completed --- which is actually built on top of ignored-patterns, so if you use that in the way familiar to current zsh users, where the ignored matches are shown if there are no unignored matches, you need the _ignored completer in your completer list.

One slightly annoying feature with _ignored is if there is only a single possible completion, since it will then be unconditionally inserted. Hardly a surprise, but it can be annoying if you really don't want that choice. There is a style single-ignored which you can set to show --- just show the single ignored match, don't insert it --- or to menu --- go to menu completion so that TAB cycles you between the completion which _ignored produced and what you originally typed. The latter gives a very natural way of handling ignored files; it's sort of saying `well, I found this but you might not like it, so hit tab again if you want to go back to what you had before'.

I said this was like _prefix, and indeed you can specify which completers are called for the _ignored completer in just the same way, by giving the completer style in the context `:completion:*:ignored:*'. That means my description has been a little over-simplified: _ignored doesn't really use the completions which were ignored before; rather, when it's called it generates a list of possibilities where the choices matched by ignore-patterns --- or internally using $fignore --- are not ignored. So it should really be called `_not_ignored', but it isn't.

_match

This and the remaining completers are utilities, which affect the main completers given above when put into the completion list rather than doing completion themselves.

The _match completer should appear *after* _complete; it is a more flexible form of the GLOB_COMPLETE option. In other words, if _complete didn't succeed, it will try to match the word on the line as a pattern, not just a fixed string, against the possible completions. To make it work like normal completion, it usually acts as if a `*' was inserted at the cursor position, even if the word already contains wildcards.

You can control the addition of `*' with the `match-original' style; the normal behaviour occurs if this is unset. If it is set to `only', the `*' is not inserted, and if it is `true', or actually any other string, it will try first without the `*', then with. For example, consider typing `setopt c*ect<TAB>' with the _match completer in use. Normally this will produce two possibilities, `correct' and `correctall'. After setting the style,

```
  zstyle ':completion::match:*' original only
```

no `*' would be inserted at the place where you hit `TAB', so that `correct' is the only possible match.

The _match completer uses the style insert-unambiguous in just the same way as does _approximate.

_all_matches

This has a similar effect to performing expansion instead of completion: all the possible completions are inserted onto the command line. However, it uses the results of ordinary contextual completion to achieve this. The normal way that the completion system achieves this is by influencing the behaviour of any subsequent completers which are called --- hence you will need to put _all_matches in the list of completers before any which you would like to have this behaviour.

You're unlikely to want to do this with every type of completion, so there are two ways of limiting its effect. First, there is the avoid-completer style: you can set this to a list of completers which should *not* insert all matches, and they will be handled normally.

Then there is the style old-matches. This forces _all_matches to use an existing list of matches, if it exists, rather than what would be generated this time round. You can set the style to only instead of true; in this case _all_matches will never apply to the completions which would be generated this time round, it will only use whatever list of completions already exists.

This can be a nuisance if applied to normal completion generation --- the usual list would never be generated, since _all_matches would just insert the non-existent list from last time --- so the manual recommends two other ways of using the completer with this style. First, you can add a condition to the use of the style:

```
  zstyle -e ':completion:*' old-matches 'reply=(${NUMERIC:-false})'
```

This returns false unless there is a non-zero numeric argument; if you type `<ESC>1` in emacs mode, or just `1` in vi mode, before completion, it will insert all the values generated by the immediately preceding completion.

Otherwise, you can bind `_all_matches` separately. This is probably the more useful; copying the manual entry:

```
  zle -C all-matches complete-word _generic
  bindkey '^Xa' all-matches
  zstyle ':completion:all-matches:*' completer _all_matches
  zstyle ':completion:all-matches:*' old-matches only
```

Here we generate ourselves a new completion based on the `complete-word` widget, called `all-matches` --- this name is arbitrary but convenient. We bind that to the keystroke `^Xa`, and give it two special styles which normal completion won't see. For the `completer` we set just `_all_matches`, and for `old-matches` we set `only`; the effect is that `^Xa` will only ever have the effect of inserting all the completions which were generated by the last completion, whatever that was --- it does not have to be an ordinary contextual completion, it may be the result of any completion widget.

`_list`

If you have this in the list of completers (at the beginning is as good as anything), then the first time you try completion, you only get a list; nothing changes, not even a common prefix is inserted. The second time, completion continues as normal. This is like typing `^D`, then tab, but using just the one key. This differs from the usual `AUTO_LIST` behaviour in that is entirely irrespective of whether the completion is ambiguous; you always get the list the first time, and it always does completion in the usual way the second time.

The `_list` completer also uses the `condition` style, which works a bit like the styles for the `_expand` completer: it must be set to one of the values corresponding to `true' for the `_list` delaying behaviour to take effect. You can test for a particular value of `$NUMERIC` or any other condition by using the `-e` option of `zstyle` when defining the style.

Finally, the boolean style `word` is also relevant. If false or unset, `_list` examines the whole line when deciding if it has changed, and hence completion should be delayed until the next keypress. If true, it just examines the current word. Note that `_list` has no knowledge of what happens between those completion calls; looking at the command line is its only resource.

`_menu`

This just implements menu completion in shell code; it should come before the `real' completion generators in the `completers` style. It ignores the `MENU_COMPLETION` option and other related options and the normal menu-completion widgets don't work well with it. However, you can copy it and write your own completers.

`_oldlist`

This completer is most useful when you are in the habit of using special completion functions, i.e. commands other than the standard completion system. It is able to hang onto an old completion list which would otherwise be replaced with a newly generated one. There are two aspects to this.

First, listing. Suppose you try to complete something from the shell history, using the command bound to `` `ESC-/' ``. For example, I typed `` `echo ma<ESC-/>' `` and got `` `max-errors' ``. At this point you might want to list the possible completions. Unfortunately, if you type `^D`, it will simply list all the usual contextual completions --- for the `echo` command, which is not handled specially, these are simply files. So it doesn't work. By putting the `_oldlist` completer into the `completers` style *before* `_complete`, it does work, because the old list of matches is kept for `^D` to use.

In this case, you can force old-listing on or off by setting the `old-list` style to `always` or `never`; usually it shows the listing for the current set of completions if that isn't already displayed, and otherwise generates the standard listing. You can even set the value of `old-list` to a list of completers which will always have their list kept in this way.

The other place where `_oldlist` is useful is in menu completion, where exactly the same problem occurs: if you generate a menu from a special command, then try to cycle through by hitting tab, completion will look for normal contextual matches instead. There's a way round this time --- use the special command key repeatedly instead of tab. This is rather tedious with multiple key sequences. Again, `_oldlist` cures this, and again you can control the behaviour with a style, `old-menu`, which takes a boolean value (it is on by default). As Orwell put it, oldlisters unbellyfeel menucomp.

Ordering completers

I've given various suggestions about the order in which completers should come in, which might be confusing. Here, therefore, is a suggested order; just miss out any completers you don't want to use:

```
  _all_matches _list _oldlist _menu _expand _complete _match
    _ignored _correct _approximate _prefix
```

Other orders are certainly possible and maybe even useful: for example, the `_all_matches` completer applies to all the completers following not listed in the `avoid-completer` style, so you might have good reason to shift it further down the list.

Here's my example of labels for completers, which I mentioned just above the list of different completers, whereby completers can be looked up under different names.

```
  zstyle ':completion:*' completer _complete _approximate:-one \
    _complete:-extended _approximate:-four
  zstyle ':completion:*:approximate-one:*' max-errors 1
  zstyle ':completion:*:complete-extended:*' \
    matcher 'r:|[.,_-]=* r:|=*'
  zstyle ':completion:*:approximate-four:*' max-errors 4
```

This tries the following in order.

1. Ordinary, no-frills completion.
2. Approximation with one error, as given by the second style.
3. Ordinary completion with extended completion turned on, as given by the third style. Sorry, this will be a black box until I talk about the `matcher` style later on; for now, you'll just have to take my word for it that this style allows the characters in the square brackets to have a wildcard in front, so ``a-b`' can complete to ``able-baker`', and so on.
4. Approximation with up to four errors, as given by the final style.

Here's a rather bogus example. You have a directory containing:

```
  foobar  fortified-badger  frightfully-barbaric
```

Actually, it's not bogus at all, since I just created one. First try ``echo foo<TAB>`'; no surprise, you get ``foobar`'. Now try completing with ``fo-b<TAB>`' after the ``echo`': basic completion fails, it gets to `_approximate:-one' and finds that it's allowed one error, so accepts the completion ``foobar`' again. Now try ``fort-ba<TAB>`'. This time nothing kicks in until the third completion, which effectively allows it to match ``fort*-ba*<TAB>`', so you see ``fortified-badger`' (no, I've never seen one myself, but they're nocturnal, you know). Finally, try ``fortfully-ba<TAB>`'; the last entry, which allows up to four errors, thoughtfully corrects ``or`' to ``righ`', and you get ``frightfully-barbaric`'. All right, the example is somewhat unhinged, but I think you can see the features are useful. If it makes you feel better, it took me four or five attempts to get the styles right for this.

6.5.2: Changing the format of listings: groups etc.

format

You can use this style if you want to find out where the completions in a completion listing come from. The most basic use is to set it for the `descriptions` tag in any completion context. It takes a string value in which `` `%d' `` should appear; this will be replaced by a description of whatever is being completed. For example, I use:

```
  zstyle ':completion:*:descriptions' format 'Completing %d'
```

and if I type `cd^D`, I see a listing like this (until I define the `group-name` style, that is):

```
  Completing external command
  Completing builtin command
  Completing shell function
  cd                 cddbsubmit         cdp                cdrecord
  cdctrl             cdecl              cdparanoia         cdswap
  cdda2wav           cdmatch            cdparanoia-yaf
  cddaslave          cdmatch.newer      cdplay
  cddbslave          cdot               cdplayer_applet
```

The descriptions at the top are related to the tag names --- usually there's a unique correspondence --- but are in a more readable form; to get the tag names, you need to use `^Xh`. You will no doubt see something different, but the point is that the completions listed are a mixture of external commands (e.g. `cdplay`), builtin commands (`cd`) and shell functions (`cdmatch`, which happens to be a leftover from old-style completion, showing you how often I clean out my function directory), and it's often quite handy to know what you have.

You can use some prompt escapes in the description, specifically those that turn on or off standout mode (`` `%S' ``, `` `%s' ``), bold text (`` `%B' ``, `` `%b' ``), and underlined text (`` `%U' ``, `` `%u' ``), to make the descriptions stand out from the completion lists.

You can set this for some other tag than `descriptions` and the format thus defined will be used only for completions of that tag.

group-name, group-order

In the `format` example just above, you may have wondered if it is possible to make the different types of completion appear separately, together with the description. You can do this using *groups*. They are also related to tags, although as you can define group names via the `group-name` style it is possible to give different names for completion in any context. However, to start off with it is easiest to give the value of the style an empty string, which means that group names are just the names of the tags. In other words,

```
  zstyle ':completion:*' group-name ''
```

assigns a different group name for each tag. Later, you can fine-tune this with more specific patterns, if you decide you want various tags to have the same group name. If no group name is defined, the group used is called `` `-default-' ``, so this is what was happening before you issued the `zstyle` command above; all matches were in that group.

The reason for groups is this: matches in the same group are shown together, matches in different groups are shown separately. So the completion list from the previous example, with both the `format` and `group-name` styles set, becomes:

```
  Completing external command
  cdctrl             cddbsubmit         cdparanoia         cdrecord
  cdda2wav           cdecl              cdparanoia-yaf
  cddaslave          cdot               cdplay
  cddbslave          cdp                cdplayer_applet
  Completing builtin command
  cd
  Completing shell function
  cdmatch                 cdmatch.newer           cdswap
```

which you may find more helpful, or you may find messier, depending on deep psychological factors outside my control.

If (and only if) you are using `group-name`, you can also use `group-order`. As its name suggests, it determines the order in which the different completion groups are displayed. It's a little like `tag-order`, which I described when tags were first introduced: the value is just a set of names of groups, in the order you want to see them. The example from the manual is relevant to the listing I just showed:

```
  zstyle ':completion:*:-command-' group-order \
      builtins functions commands
```

--- remember that the `` `-command-' `` context is used when the names of commands, rather than their arguments, are being completed. Not surprisingly, that listing now becomes:

```
  Completing builtin command
  cd
  Completing shell function
  cdmatch                 cdmatch.newer           cdswap
  Completing external command
  cdctrl            cddbsubmit        cdparanoia        cdrecord
  cdda2wav          cdecl             cdparanoia-yaf
  cddaslave         cdot              cdplay
  cddbslave         cdp               cdplayer_applet
```

and if you investigate the tags available by using `^Xh`, you'll see that there are others such as aliases whose order we haven't defined. These appear after the ones for which you have defined the order and in some order decided by the function which generated the matches.

`tag-order`

As I already said, I've already described this, but it's here again for completeness.

`verbose`, `auto-description`

These are relatives of `format` as they add helpful messages to the listing. If `verbose` is true, the function generating the matches may, at its discretion, decide to show more information about them. The most common case is when describing options; the standard function `_describe` that handles descriptions for a whole lot of options tests the `verbose` style and will print information about the options it is completing.

You can also set the string style `auto-description`; it too is useful for options, in the case that they don't have a special description, but they do have a single following argument, which completion already knows about. Then the description of the argument for verbose printing will be available as `` `%d' `` in `auto-describe`, so that something like the manual recommendation `` `specify: %d' `` will document the option itself. So if a command takes `` `-o <output-file>' `` and the argument has the description `` `output file' ``, the `` `-o' ``, when it appears as a possible completion, will have the description `` `specify: output file' `` if it does not have its own description. In fact, most options recognized by the standard completion functions already have their own descriptions supplied, and this is more subtlety than most people will probably need.

`list-colors`

This is used to display lists of matches for files in different colours depending on the file type. It is based on the syntax of the `$LS_COLORS` environment variable, used by the GNU version of `ls`. You will need a terminal which is capable of displaying colour such as a colour xterm, and should make sure the `zsh/complist` library is loaded, (it should be automatically if you are using menu selection set up with the `menu` style, or if you use this style). But you can make sure explicitly:

```
  zmodload -i zsh/complist
```

The `-i` keeps it quiet if the module was already loaded. To install a standard set of default colours, you can use:

```
  zstyle ':completion:*' list-colors ''
```

--- note the use of the `` `default' `` tag --- since a null string sets the value to the default.

If that's not good enough for you, here are some more detailed instructions. The parameter `$ZLS_COLORS` is the lowest-level part of the system used by `zsh/complist`. There is a simple builtin default, while having the style set to the empty string is equivalent to:

```
  ZLS_COLORS="no=00:fi=00:di=01;34:ln=01;36:\
  pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:\
  ex=01;32:lc=\e[:rm=m:tc=00:sp=00:ma=07:hi=00:du=00
```

It has essentially the same format as `$LS_COLORS`, and indeed you can get a more useful set of values by using the `dircolors` command which comes with `ls`:

```
  ZLS_COLORS="no=00:fi=00:di=01;34:ln=01;36:\
  pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:\
  or=40;31;01:ex=01;32:*.tar=01;31:*.tgz=01;31:\
  *.arj=01;31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:\
  *.z=01;31:*.Z=01;31:*.gz=01;31:*.deb=01;31:\
  *.jpg=01;35:*.gif=01;35:*.bmp=01;35:*.ppm=01;35:\
  *.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:\
  *.mpg=01;37:*.avi=01;37:*.gl=01;37:*.dl=01;37:"
```

You should see the manual for the `zsh/complist` module for details, but note in particular the addition of the type `` `ma' ``, which specifies how the current match in menu selection is displayed. The default for that is to use standout mode --- the same effect as the sequence `%S` in a prompt, which you can display with `` `print -P %Sfoo' ``.

However, you need to define the style directly, since the completion always uses that to set `$ZLS_COLORS`; otherwise it doesn't know whether the value it has found has come from the user or is a previous value taken from some style. That takes this format:

```
  zstyle ':completion:*' list-colors "no=00" "fi=00" ...
```

You can use an already defined `$LS_COLORS`:

```
  zstyle ':completion:*' list-colors ${(s.:.)LS_COLORS}
```

(which splits the parameter to an array on colons) as `$LS_COLORS` is still useful for `ls`, even though it's not worth setting `$ZLS_COLORS` directly. This should mean GNU ls and zsh produce similar-looking lists.

There are some special effects allowed. You can use patterns to tell how filenames are matched: that's part of the default behaviour, in fact, for example '*.tar=01;31' forces tar files to be coloured red. In that case, you are limited to `` `*' `` followed by a string. However, there's a way of specifying colouring for any match, not just files, and for any pattern: use `=<pat>=<col>`. Here are two ways of getting jobs coloured red in process listings for the `` `kill' `` command.

```
  zstyle ':completion:*:*:kill:*' list-colors '=%*=01;31'
```

This uses the method just described; jobs begin with `` `%' ``.

```
  zstyle ':completion:*:*:kill:*:jobs' list-colors 'no=01;31'
```

This uses the tag, rather than the pattern, to match the jobs lines. It has various advantages. Because you are using the tag, it's much easier to alter this for all commands using jobs, not just kill --- just miss out `` `kill' `` from the string. That wasn't practical with the other method because it would have matched too many other things you didn't want. You're not dependent on using a particular pattern, either. And finally, if you try it with a `format' description you'll see that that gets the colour, too, since it matched the correct tag. Note the use of the `no' to specify that this is to apply for a normal match; the other two-letter codes for file types aren't useful here.

However, there is one even more special effect you can use with the general pattern form. By turning on `backreferences' with `(#b)' inside the pattern, parentheses are active and the bits they match can be coloured separately. You do this by extending the list of colours, each code preceded by an `=' sign, and the extra elements will be used to colour what the parenthesis matched. Here's another example for `kill', which turns the process number red, but leaves the rest alone.

```
  zstyle ':completion:*:*:kill:*:processes' list-colors \
    '=(#b) #([0-9]#)*=0=01;31'
```

The hieroglyphics are extended globbing patterns. You should note that the EXTENDED_GLOB option is always on inside styles --- it's required for the `#b' to take effect. In particular, `#' means `zero or more repetitions of the previous bit of the pattern' with extended glob patterns; see the globbing manual page for full details.

ignored-patterns

Many shells, including zsh, have a parameter $fignore, which gives a list of suffixes; filenames ending in any of these are not to be used in completion. A typical value is:

```
  fignore=(.o \~ .dvi)
```

so that normal file completion will not produce object files, EMACS backup files, or TeX DVI files.

The ignored-patterns style is an extension of this. It takes an array value, like fignore, but with various differences. Firstly, these values are patterns which should match the *whole* value to be completed, including prefixes (such as the directory part of a filename) as well as suffixes. Secondly, they apply to *all* completions, not just files, since you can use the style mechanism to tune it to apply wherever you want, down to particular tags.

Hence you can replace the use of $fignore above with the following:

```
  zstyle ':completion:*:files' ignored-patterns '*?.o' '*?~' '*?.dvi'
```

for completion contexts where the tag `files' is in use. The extra `?'s are because $fignore was careful only to apply to real suffixes, i.e. strings which had something in front of them, and the `?' forces there to be at least one character present.

Actually, this isn't quite the same as $fignore, since there are other file tags than files; apart from those for directories, which you've already met, there are globbed-files and all-files. The former is for cases where a pattern is specified by the completion function, for example `*.dvi' for files following the command name dvips. These don't use this style, because the pattern was already sufficiently specified. This follows the behaviour for $fignore in the old completion system. Another slight difference, as I said above when discussing the _ignored completer, is that you get to choose whether you want to see those ignored files if the normal completions fail, by having _ignored in the completer list or not.

The other tag, all-files, applies when a globbed-files tag failed, and says any old file is good enough in that case; you can arrange how this happens with the tag-order style. In this example,

```
  zstyle ':completion:*:*:dvips:argument*' \
    tag-order globbed-files all-files
```

is enough to say that you want to see all files if no files were produced from the pattern, i.e. if there were no `*.dvi' files in the directory. Finally the point of this ramble: as the all-files tag is separate from the files tag, in this case you really would see all files (except for those beginning with a `.', as usual). You might find this useful, but you can easily make the `all-files` tag behave the same way as `files`:

```
  zstyle ':completion:*:(all-|)files' ignored-patterns ...
```

Here's the example of using tag labels I promised earlier; it's simply taken from the manual. To refresh your memory: tag labels are a way of saying that tags should be looked up under a different name. Here we'll do:

```
  zstyle ':completion:*:*:-command-:*' tag-order 'functions:-non-comp'
```

This applies in command position, from the special `-command-' context, the place where functions occur most often, along with other types of command which have their own tags. This says that when functions are first looked up, they are to be looked up with the name `functions-non-comp' --- remember that with a hyphen as the first character of the label part, the bit after the colon, the `functions` tag name itself, the bit before the colon, is to be stuck in front to give the full label name `functions-non-comp'. We can use it as follows:

```
  zstyle ':completion:*:functions-non-comp' ignored-patterns '_*'
```

In the context of this tag label, we have told completion to ignore any patterns --- i.e. any function names --- beginning with an underscore. What happens is this: when we try completion in command position, `tag-order` is looked up and finds we want to try functions first, but under the name `functions-non-comp`; this completes functions apart from ones beginning with an underscore (presumably completion functions you don't want to run interactively). Since `tag-order` normally tries all the other tags, unless it was told not to, in this case all the normal command completions will appear, including functions under their normal tag name, so this just acts as a sort of filter for the first attempt at completion. This is typically what tag labels are intended for --- though maybe you can think up a lot of other uses, since the idea is quite powerful, being backed up by the style mechanism.

You way wonder why you would want to ignore such functions at this point. After all, you're only likely to be doing completion when you've already typed the first character, which either is `_' or it isn't. It becomes useful with correction and approximation --- particularly since many completion functions are similar to the names of the commands for which they handle completion. You don't want to be offered `_zmodload' as a completion if you really want `zmodload'. The combination of labels and ignored patterns does this for you.

You can generalise this using another feature: tags can actually be patterns, which I mentioned but didn't demonstrate. Here's a more sophisticated version of the previous example, adapted from the manual:

```
  zstyle ':completion:*:*:-command-:*' tag-order \
  'functions:-non-comp:non-completion\ functions *' functions
```

It's enhanced so that completion tries all other possible tags at the same time as the labelled `functions`. However, it only ever tries a tag once at each step, so the `*' doesn't put back `functions` as you might expect --- that's still tried under the label `functions-non-comp', and the `ignored-patterns` style we set will still work. In the final word, we try all possible functions, so that those beginning with an underscore will be restored.

Use of the `_ignored' completer can allow you to play tricks without having to label your tags:

```
  zstyle ':completion:*' completer _complete _ignored
  zstyle ':completion:*:functions' ignored-patterns '_*'
```

Now anywhere the `functions` tag is valid, functions matching `_*' aren't shown until completion reaches the `_ignored' in the completer list. Of course, you should manipulate the completer list the way you want; this just shows the bare bones.

`prefix-hidden`, `prefix-needed`

You will know that when the shell lists matches for files, the directory part is removed. The boolean style `prefix-hidden` extends this idea to various other types of matches. The prefixes referred to are not just any old common prefix to matches, but only some places defined in the completion system: the `-` prefix to options, the `` `%' `` prefix to jobs, the `-` or `+` prefix to directory stack entries are the most commonly used.

The `prefix-needed` applies not to listings, but instead to what the user types on the command line. It says that matches will only be generated if the user has typed the prefix common to them. It applies on broadly the same occasions as `prefix-hidden`.

`list-packed`, `list-rows-first`, `accept-exact`, `last-prompt`, `menu`

The first two of these have already been introduced, and correspond to the `LIST_PACKED` and `LIST_ROWS_FIRST` options. The `accept-exact` and `last-prompt` styles correspond essentially to the `REC_EXACT` and `ALWAYS_LAST_PROMPT` options in the same way.

The style `menu` roughly corresponds to the `MENU_COMPLETE` option, but there is also the business of deciding whether to use menu selection, as described above. These two uses don't interfere with each other --- except that, as I explained, menu completion must be started to use menu selection --- so a value like `` `true select=6' `` is valid; it turns on menu completion for the context, and also activates menu selection if there are at least 6 choices.

There are some other, slightly more obscure, choices for `menu`:

`yes=`*num*
: turn on menu completion only if there are at least *num* matches;

`no=`*num*
: turn off menu completion if there are as many as *num* matches;

`yes=long`
: turn on menu completion if the list does not fit on the screen, and completion was attempted;

`yes=long-list`
: the same, but do it even if listing, not completion, was attempted;

`select=long`
: like `yes=long`, but this time turn on menu selection, too;

`select=long-list`
: like `yes=long-list`, but turn on menu selection, too.

In case your eyes glazed over before the end, here's a full description of the last one, `select=long-list`, which is quite useful: if you are attempting completion or even just listing completions, and the list of matches would be too long to fit on the screen, then menu selection is turned on, so that you can use the cursor keys (and other selection keys) to move up and down the list. Generally, the above possibilities can be combined, unless the combined effect wouldn't work.

As always, `yes` and `true` are equivalent, as are `no` and `false`. It just hurts the eyes of programmers to read something which appears to assign a value to `true`.

`hidden`

This is a little obscure for most users. Its context should be restricted to specific tags; any corresponding matches will not be shown in completion listings, but will be available for inserting into the command line. If its value is `` `true' ``, then the description for the tag may still appear; if the value is `` `all' ``, even that is suppressed. If you don't want the completions even to be available for insertion, use the `tag-order` style.

6.5.3: Styles affecting particular completions

The styles listed here are for use only with certain completions as noted. I have not included the styles used by particular completers, which are described with the completer in question in the subsection `**Specifying completers and their options**'. I have also not described styles used only in separate widgets that do completion; the relevant information is all together in the next section.

Filenames (1): patterns: `file-patterns`

It was explained above for the `tag-order` style that when a function uses pattern matching to generate file completions, such as all `*.ps` files or all `*.gz` files, the three tags `globbed-files`, `directories` and `all-files` are tried, in that order.

The `file-patterns` style allows you to specify a pattern to override whatever would be completed, even in what would otherwise be a simple file completion with no pattern. Since this can easily get out of hand, the best way of using this style is to make sure that you specify it for a narrowly enough defined context. In particular, you probably want to restrict it to completions for a single command and for a particular one of the tags usually applying to files. As always, you can use `^Xh` to find out what the context is. It has a labelling mechanism --- you can specify a tag with a pattern for use in looking up other styles. Hence `` `*.o:object-files' `` gives a pattern `` `*.o' `` and a tag name `` `object-files' `` by which to refer to these.

The patterns you specify are tried in order; you don't need to use `tag-order`. In fact `file-patterns` replicates its behaviour in that you can put patterns in the same word to say they should be tried together, before going on to the pattern(s) in the next word. Also, you can give a description after a second colon in the same way. Indeed, since `file-patterns` gets its hands on the tags first, any ordering defined there can't be overridden by `tag-order`.

So, for example, after

```
  zstyle ':completion:*:*:foo:*:*' file-patterns \
    '*.yo:yodl-files:yodl\ files *(-/):directories'
```

the command named `` `foo' `` will complete files ending in `` `.yo' ``, as well as directories. For once, you don't have to change the completer to alter what's completed: `` `foo' `` isn't specially handled, so it causes default completion, and that means completing files, so that `file-patterns` is active anyway.

Here's a slightly enhanced example; it shows how `file-patterns` can be used instead of `tag-order` to offer the tags in the order you want.

```
  zstyle ':completion:*:*:foo:*:*' file-patterns \
    '*.yo:yodl-files:yodl\ files' '*(-/):directories:directories' \
    '^*.yo(-^/):other-files:other\ files'
```

Completion will first try to show you only `` `.yo' `` files, if there are any; otherwise it will show you directories, if there are any; otherwise it will show you any other files: `` `^*.yo(-^/)' `` is an extended glob to match any file which doesn't end in `` `.yo' `` and which isn't a directory and doesn't link to a directory. As always, you can cycle through the sets of possibilities using the `` `_next_tag' `` completion command.

Note that `file-patterns` is an exception to the general rule that styles don't determine *which* tags are called only *where* they're called, or what their behaviour is: this time, you actually get to specify the set of tags which will be used. This means it doesn't use the the standard file tags (unless you use those names yourself, of course), just `` `files' `` if you don't specify one. Hence it's good style to add the tags, following colons, although it'll work without.

Another thing to watch out for is that if there is already a completion which handles a file type --- for example, if we had tried to alter the effect of file completion for the `` `yodl' `` command instead of the fictitious `` `foo' `` --- the results may well not be quite what you want.

Another feature is that `%p' in the pattern inserts the pattern which would usually be used. That means that the following is essentially the same as what file completion normally does:

```
  zstyle ':completion:*' file-patterns '%p:globbed-files' \
    '*(-/):directories' '*:all-files'
```

You can turn completion for a command that usually doesn't use a pattern into one that does. Another example taken from the manual:

```
  zstyle ':completion:*:*:rm:*:globbed-files' file-patterns \
    '*.o:object-files' '%p:all-files'
```

So if there are any `*.o` files around, completion for `rm` will just complete those, even if arguments to `rm` are otherwise found by default file completion (which they usually are). The `%p` will use whatever file completion normally would have; probably any file at all. You can change this, if you like; there may be files you don't ever want automatically completed after `rm`.

Remember that using explicit patterns overrides the effect of `$fignore`; this is obviously useful with `rm`, since the files you want to delete are often those you usually don't want to complete.

Filenames (2): paths: `ambiguous`, `expand`, `file-sort`, `special-dirs`, `ignore-parents`, `list-suffixes`, `squeeze-slashes`

Filename completion is powerful enough to complete all parts of a path at once, for example `` `/h/p/z' `` will complete to `` `/home/pws/zsh' ``. This can cause problems when the match is ambiguous; since several components of the path may well be ambiguous, how much should the completion system complete, and where should it leave the cursor? This facility is associated with all these styles affecting filenames.

With ordinary completion, the usual answer is that the completion is halted as soon as a path component matches more than one possibility, and the cursor is moved to that point, with the remainder of the string left unaltered. With menu completion, you can simply cycle through the possibilities with the cursor moved to the end as usual. If you set the style `ambiguous`, then the system will leave the cursor at the point of the first ambiguity even if menu completion is in use. Note that this is always used with the `` `paths' `` tag, i.e. the context ends in `` `...:paths' ``.

The style `expand` is similar and is also applied with the `` `paths' `` tag. It can include either or both of the strings `prefix` and `suffix`. Be careful when setting both --- they have to be separate words, for example

```
  zstyle ':completion:*' expand prefix suffix
```

Don't put quotes around `` `prefix suffix' `` as it won't work.

With `prefix`, `expand` tells the completion system always to expand unambiguous prefixes in a path (such as `` `/u/i' `` to `` `/usr/in' ``, which matches both `/usr/include` and `/usr/info`) --- even if the remainder of the string on the command line doesn't match any file. So this expansion will now happen even if you try this on `` `/u/i/ALoadOfOldCodswallop' ``, which it otherwise wouldn't.

Including `suffix` in the value of `expand` extends path completion in another way: it allows extra unambiguous parts to be added even after the first ambiguous one. So if `` `/home/p/.pr' `` would match `` `/home/pws/.procmailrc' `` or `` `/home/patricia/.procmailrc' ``, and nothing else, the last word would be expanded. Set up like this, you will always get the longest unambiguous match for all parts of the path.

In older versions of the completion system, `suffix` wasn't used if you had menu completion active by default, although it was if menu completion was only started by the `AUTO_MENU` option. However, in recent versions, the setting is always respected. This means that setting the `expand` style to include the value `suffix` allows menu completion to cycle through all possible completions, as if there were a `` `*' `` after each part of the path, so `` `/u/i/k' `` will offer all matches for `` `/u*/i*/k*' ``.

The `file-sort` style allows files to be sorted in a way other than by alphabetical order: sorting applies both to the list of files, and to the order in which menu completion presents them. The value should include one of the following: \``size`', \``links`', \``modification`' (same as \``time`', \``date`'), \``access`', \``inode`' (same as \``change`'). These pick the obvious properties for sorting: file size, number of hard links, modification time, access time, inode change time. You can also add the string \``reverse`' to the value, which reverses the order. In this case the tag is always \``files`'.

The `special-dirs` style controls completion of the special directories \``.`' and \``..`'. Given that you usually need to type an initial dot to complete anything at all beginning with one, the idea of \`completing' \``.`' is a little odd; it simply means that the directory is accepted when the completion is started on it. You can set the style to `true` to allow completion to both of the two, or to \``..`' to complete \``..`' but not \``.`'. Like `ambiguous`, this is used with the tag set to \``paths`'.

The style `ignore-parents` is used with the `files` tag, since it applies to paths, but not necessarily completion of multiple path names at once; it can be used when completing just the last element. There are two main uses, which can be combined. The first case is to include the string \``parent`' in the style. This means that when you complete after (say) `foo/../`, the string `foo` won't appear as a choice, since it already appeared in the string. Secondly, you can include \``pwd`' in the value; this means don't complete the current working directory after \``../`' --- you can see the sense in that: if you wanted to complete there, you wouldn't have typed the \``..`' to get out if it.

Actually, the function performs both those tests on the directories in question even if the string \``..`' itself hasn't been typed. That might be more confusing, and you can make sure that the tests for `parent` and `pwd` are only made when you typed the \``..`' by including a \``..`' in the style's value. Finally, you can include the string \``directory`' in the values: that means the tests will only be performed when directories are being completed, while if some other sort of file, or any file, can be completed, the special behaviour doesn't occur. You may have to read that through a couple of times before deciding if you need it or not.

Next, there is `list-suffixes`. It applies when expanding out earlier parts of the filename path, not just the last part. In this case, it is possible that early parts of the path were ambiguous. Normally completion stops at the point where it finds the ambiguity, and leaves the rest of the path alone. When `list-suffixes` is set, it will list all the possible values of all ambiguous components from the point of ambiguity onward.

Lastly, there is the style `squeeze-slashes`. This is rather simpler. You probably already know that in a UNIX filename multiple slashes are treated just like a single slash (with a few minor exceptions on some systems). However, path completion usually assumes that multiple slashes mean multiple directories to be completed: \``//termc`' completes to \``/etc/termcap`' because of this rule. If you want to stick with the ordinary UNIX rule you can set `squeeze-slashes` to `true`. Then in this example only files in the root directory will be completed.

Processes: `command`, `insert-ids`

Some functions, such as `kill`, take process IDs (i.e. numbers) as arguments. These can be completed by using the `ps` command to generate the process numbers. The `command` style allows you to specify which arguments are to be passed to `ps` to generate the numbers; it is simply `eval`'d to generate the command line. For example, if you are root and want to have all processes as possible completions, you might use \``-e`', for many modern systems, or \``ax`', for older BSD-like systems. The completion system tries to find a column which is headed \``PID`' or \``pid`' (or even \``Pid`', in fact) to use for the process IDs; if it doesn't find one, it just uses the first column.

The default is not to use any arguments; most variants of `ps` will then just show you interactive processes from your current session. To show all your own processes on a modern system, you can probably use the value \``ps -u$USER`' for the style --- remembering to put this in single quotes. Clearly, you need to make sure the context is narrow enough to avoid unexpectedly calling odd commands.

You can make the value begin with a hyphen, then the usual command line will put afterward and the hyphen removed. The suggested use for this is adding \``command`' or \``builtin`' to make sure the right version of a command is called.

The completion system allows you to type the name of a command, for example `emacs', which will be converted to a PID. Note that this is different from a job name beginning with `%'; in this case, any command listed by ps, given the setting of the command style, can be used. Obviously, command names can be ambiguous, unlike the process IDs themselves, so the names are usually converted immediately to PIDs; if the name could refer to more than one process, you get a menu of possible PIDs.

The style insert-ids allows the completion system to keep using the names rather than the PIDs. If it is set to single, the name will be retained until you type enough to identify a particular process. If it is set to true (or anything else but menu, actually), menu completion is delayed until you have typed a string longer than the common prefix of the PIDs. This is intended to be similar to completion's usual logic --- don't do anything which gets rid of information supplied by the user --- so is probably more useful in practice than it sounds.

Job control: numbers

Builtin functions that take process IDs usually also take job specifications, strings beginning with `%' and followed either by a small number or a string. The style numbers determines how these are completed. By default, the completion system will try to complete an unambiguous string from the name of the job. If you set numbers to true, it will instead complete the job number --- though the listing will still show the full information --- and if you set it to a number, it will only use that many words of the job name, and switch to using numbers if those are not unique. In other words, if you set it to `1' and you have two jobs `vi foo' and `vi bar', then they will complete as `%1' and `%2' (or maybe other numbers) since the first words are the same.

Note also that prefix-needed applies here; if it is set, you need to type the `%' to complete jobs rather than processes.

System information: users, groups, hosts etc.

There are many occasions where you complete the names of users on the system, groups on the system (not to be confused with completion groups), names of other hosts you connect to via the network, and ports, which are essentially the names of internet services available on another host such as nntp or smtp.

By default, the completion system will query the usual system files to find the names of users, groups, hosts and ports, though in the final case it will only look in the file `/etc/hosts', which often includes only a very small number of not necessarily very useful hosts. It is possible to tell the completion system always to use a specified set by setting the appropriate style --- users, groups, hosts, ports --- to the set of possibilities you want. This is nearly always useful with hosts, and on some systems you may find it takes an inordinate amount of time for the system to query the database for groups and users, so you may want to specify a subset containing just those you use most often.

There are also three sets of combinations: hosts-ports, hosts-ports-users and users-hosts. These are used for commands which can take both or all three arguments. Currently, the command socket uses hosts-ports, telnet uses hosts-ports-users, while the style users-hosts is used by remote login commands such as rsh and ssh, and anywhere the form `user@host' is valid.

The last is probably the most useful, so I'll illustrate that. By setting:

```
  zstyle ':completion:*' users-hosts \
    pws:foo.bar.uk peters@frond.grub.uk
```

you tell rsh and friends the possible user/host combinations. Note that for the separator you can use either `:', as usual inside the completion system, or `@', which is more natural in this particular case. If you type `rsh -l ', a username is expected and either pws or peters will be completed. Suppose you picked pws; then for the next argument, which should be a host, the system now knows that it must be foo.bar.uk, since the username for the other host doesn't match.

If you don't need that much control, completion for all these commands will survive on just the basic `` `hosts' ``, `` `users' ``, etc. styles; it simply won't be as clever in recognising particular combinations. In fact, even if you set the combined styles, anything that doesn't match will be looked up in the corresponding basic style, so you can't lose, in principle.

The other combined styles work in exactly the same way; just set the values separated by colons or `` `@' ``, it doesn't matter which.

URLs for web browsers

Completion for URLs is done by setting a parallel path somewhere on your local machine. The `urls` style specifies the top directory for this. For example, to complete the URL `http://zsh.org/`, you need to make a set of subdirectories of the `path` directory `http/zsh.org/`. You can extend this for however many levels of directory you need; as you would expect, if the last object is a file rather than a directory you should create it with `` `touch' `` rather than `` `mkdir' ``. The style will always use the tag `` `urls' `` for this purpose, i.e. the context always matches `` `:completion:*:urls' ``. This is a neat way of using the ordinary filing system for doing the dirty work of turning URLs into components. Arguably the system should be able to scan your browser's bookmarks file, but currently it won't; there is, however, a tool provided with the shell distribution in `Misc/make-zsh-urls` which should be able to help --- ask your system administrators about this if it isn't installed, I'm sure they'll be delighted to help.

If you only have a few URLs you want to complete, you can use one of two simpler forms for the `urls` style. First, if the value of the style contains more than one word, the values are used directly as the URLs to be completed, e.g.:

```
  zstyle ':completion:*:urls' urls \
      http://www.foo.org/ ftp://ftp.bar.net
```

Alternatively, you can set the `urls` style to the name of a normal file, which contains the URLs to complete separated by white space or newlines.

Note that many modern browsers allow you to miss out an initial `` `http://' ``, and that lots of pseudo-URLs appear in newspapers and advertisements without it. The completion system needs it, however.

There is a better way when the web pages actually happen to be hosted on a system whose directories you can access directly. Set the `local` style to an array of three strings: a hostname to be considered local (you can only give one per context), the directory corresponding to the root of the files, and the directory where a user places their own web pages, relative to their home directory. For example, if your home page is usually retrieved as `http://www.footling.com/`, and that looks for the index file (often called `index.html`) in the directory `/usr/local/www/files`, and your own web pages live under `` `~/www' ``, then you would set

```
  zstyle ':completion:*:urls' local \
      www.footling.com /usr/local/www/files www
```

and when you type `` `lynx http://www.footling.com/' ``, all the rest will be completed automatically.

The X files

There is another use for the `path` style with the tag `` `colors' ``: it gives the path to a file which contains a list of colour names understood by the X-windows system, usually in file named `` `rgb.txt' ``. This is used in such contexts as `` `xsetroot -solid ' ``, which completes the name of a colour to set your root window (wallpaper) to. It may be that the default value works on your system without your needing to set this.

6.6: Command widgets

6.6.1: `_complete_help`

You've already met this, usually bound to \``^Xh`' unless you already had that bound when completion started up (in which case you should pick your own binding and use \``bindkey`'), but don't forget it, since it's by far the easiest way of finding out what context to use for setting particular styles.

6.6.2: `_correct_word`, `_correct_filename`, `_expand_word`

The first and last of these have been mentioned in describing the related completers: `_correct_word`, usually bound to `^Xc`, calls the `_correct` completer directly to perform spelling correction on the current word, and `_expand_word`, usually bound to `^Xe`, does the same with the `_expand` completer. The contexts being \``:completion:complete-word`' and \``:completion:expand-word`' respectively, so that they can be distinguished in styles from the ordinary use of the completer. If you want the same styles to be used in both contexts, but not others, you should define them for patterns beginning \``:completion:complete(|-word)...`'.

The middle one simply corrects filenames, regardless of the completion context. Unlike the others, it can also be called as an ordinary function: pass it an argument, and it will print out the possible corrections. It does this because it bypasses most of the usual completion system. Probably you won't often need it, but it is usually bound to \``^XC`' (note the capital \``C`').

6.6.3: `_history_complete_word`

This is usually bound to \``<ESC-/>`' for completing back in the history, and \``<ESC-,>`' for completing forward --- this will automatically turn on menu completion, temporarily if you don't normally have that set, to cycle through the matches. It will complete words from the history list, starting with the most recent. Hence

```
  touch supercalifragilisticexpialidocious
  cat sup<ESC-/>
```

will save you quite a bit of typing --- although in this particular case, you can use \``<ESC-.>`' to insert the last word of the previous command.

Various styles are available. You can set the \``stop`' style which makes it stop once before cycling past the end (or beginning) of the history list, telling you that the end was reached.

You can also set the \``list`' style to force matches to be listed, the \``sort`' style to sort matches in alphabetical order instead of by their age in the history list, and the \``remove-all-dups`' style, which ensures that each match only occurs once in the completion list --- normally consecutive identical matches are removed, but the code does not bother searching for identical matches elsewhere in the list of possibilities. Finally, the `range` style is supported via the `_history` completer, which does the work. This style restricts the number of history words to be searched for matches and is most useful if your history list is large. Setting it to a number *n* specifies that only the last *n* history words should be searched for possible matches. Alternatively, it can be a value of the form \`*max*:*slice*', in which case it will search through the last *slice* history words for matches, and only if it doesn't find any, the *slice* words before that; *max* gives an overall limit on the maximum number of words to search through.

6.6.4: `_most_recent_file`

This function is normally bound to \``^Xm`'. It simply completes the most recently modified file that matches what's on the line already. Any pattern characters in the existing string are active, so this is a cross between expansion and completion. You can also give it a numeric prefix to show the `N`th most recently modified file that matches the pattern.

By the way, you can actually do the same by setting appropriate styles, without any new functions. The trick is to persuade the system to use the normal `_files` completer with the `file-sort` style. By restricting the use of the styles to the context of the widget --- which is simply the `_generic` completer described above:

```
  zstyle ':completion:(match-word|most-recent-file):*' \
      match-original both
  zstyle ':completion:most-recent-file::::' completer \
      _menu _files _match
  zstyle ':completion:most-recent-file:*' file-sort modification
  zstyle ':completion:most-recent-file:*' file-patterns \
      '*(.):normal\ files'
  zstyle ':completion:most-recent-file:*' hidden true
  zstyle ':completion:most-recent-file:*:descriptions' format ''
  bindkey '^Xm' most-recent-file
  zle -C most-recent-file menu-complete _generic
```

It may not be obvious how this works, so here's a blow by blow account if you are interested. (It works even if you aren't interested, however.)

- The `` `zle -C' `` defines a widget which does menu completion, and behaves like ordinary completion (that's what `_generic` is for) except that the context uses the name of the widget we define.
- When we invoke the widget, the system uses the `completer` style to decide what completions to perform. This instructs it: use menu completion, complete files, use pattern matching if the completion so far didn't work.
- First, `_menu` comes along; it actually does nothing more than tell the system to use menu completion.
- Then `_files` generates a list of files. This uses the `file-sort` and `file-patterns` styles defined for the `most-recent-file` context. They produce a set of files in modification time order, and include only regular files (so not directories, symlinks, device files and so on).
- If that failed, the `_match` style allows the word on the command line to be treated as a pattern; for example, `*.c` to complete the most recent C source file. This uses the `match-original` style; the setting tells it that it should try first without adding an extra `` `*' `` for matching (this is what we want for the case where we already have a complete pattern like `*.c`), and if that fails, add a `*` at the end and try again.
- The `hidden` style means that the matches aren't listed; all that happens is the first is inserted on the line. The setting for the `format` tag similarly simplifies the display in this case by removing verbose descriptions.
- The net result is the first step of a menu completion: insert the first matched file (the most recently modified) onto the line. This is exactly what you want. Note, however, that as we are in menu completion you can keep on hitting `^xm` and the shell will cycle through the matches, which here gives you files that are progressively less recently modified.

Omit the `file-patterns` line if you don't want the match restricted to regular files (I sometimes need the most recently modified directory, but often it's irrelevant). The whole version using styles comes from Oliver Kiddle, who recommends using `_generic` in this way any time you want to generate a widget from a specific completion such as `_files`. There is a brief section on `_generic` below.

6.6.5: `_next_tags`

This is a very neat way of getting round the order of tags just with a key sequence. An example is the best way of showing it; it's bound by default to the key sequence `` `^Xn' ``.

```
  % tex ^D
  Completing TeX or LaTeX file
  bar.tex   foo.tex   guff.tex
```

Our file is not in that directory, but by default we don't get to see the directory if there was a file that matched the pattern --- here `` `*.tex' ``. (This will actually change in 4.1, since most people don't know about `_next_tags` but do know about directories, but you can still cycle through the different sets of tags.) You can set the `tag-order` style to alter whether they appear at the same time, but `_next_tags` lets you do this very simply. Just hit `^Xn`. You're now looking at

```
Completing TeX or LaTeX file
dir1/  dir2/  dir3/
```

and if you carry on hitting `^Xn` you will get to all files, and then you will be taken back to the `.tex` files again. (Where our file actually is, is left as an exercise for the reader.)

Of course this works with any set of tags whatsover; it simply has the effect of cycling you around the tag order.

6.6.6: `_bash_completions`

This function provides compatibility with a set of completion bindings in bash, in which escape followed by one of the following characters causes a certain type of (non-contextual) completion: `` `!' ``, command names; `` `$' ``, environment variables; `` `@' ``, host names; `` `/' ``, filenames, and `` `~' `` user names. `` `^X' `` followed by the same characters causes the possible completion to be listed. This function decides by examining its own binding which of those it should be doing, then calls the appropriate completion function. If you want to use it for all those possible bindings, you need to issue the right statements in your `.zshrc`, since only the bindings with `` `~' `` are set up by default to avoid clashes. This will do it:

```
for key in '!' '$' '@' '/'; do
  bindkey "\e$key" _bash_complete-word
  bindkey "^X$key" _bash_list-choices
done
```

Unlike most widgets, which are tied to functions of the same name to minimize confusion, the function `_bash_completions` is actually called under the names of the two different widgets shown in that code so as to be able to implement both completion and listing behaviour.

6.6.7: `_read_comp`

This function, usually bound to `` `^X^R' ``, does on-the-fly completion. When you call it, it prompts for you to enter a type of completion; usually this will be the name of a completion function with the required arguments. Thus it's not much use unless you already have some fairly in-depth knowledge of how the system is set up. For example, try it, then enter `` `_files -/' ``, which generates directories. There is a rudimentary completion for the function names built into it.

The next time you start it up, it will produce the same type of completion. You need to give it a numeric prefix to tell it to prompt for a different sort.

6.6.8: `_generic`

Rather than being directly bound, like the others, this widget gives you a way of creating your own special completions. You define it as a widget and bind it as if it were any completion function:

```
zle -C foo complete-word _generic
bindkey '<keys>' foo
```

Now the keys bound will perform ordinary contextual completion, but any styles will be looked up with the command context `` `foo' ``. So you can give it its own set of completers:

```
zstyle ':completion:foo:*' completer _expand
```

and, indeed, give it special values for any style you like. To put it another way, you've now got a complete, separate copy of the completion system where the only difference is the extra word in the context.

Good example of the use of this function were given above in the descriptions of `_all_matches` and `_most_recent_file`.

6.6.9: `predict-on`, `incremental-complete-word`

These are not really complete commands at all in the strict sense, they are normal editing commands which happen to have the effect of completion. This means that they are not part of the completion system, and though they are installed with other shell functions they will not automatically be loaded. You will therefore need an explicit \``autoload -U predict-on`', etc. --- remember that the \``-U`' prevents the functions from expanding any of your own aliases when they are read in --- as well as an explicit \``bindkey`' command to bind each function, and a \``zle -N`' statement to tell the line editor that the function is to be regarded as an editing widget. The `predict-on` file, when loaded, actually defines two functions, `predict-on` and `predict-off`, both of which need to be defined and bound for them to work. So to use all of these,

```
  autoload -U incremental-complete-word predict-on
  zle -N incremental-complete-word
  zle -N predict-on
  zle -N predict-off
  bindkey '^Xi' incremental-complete-word
  bindkey '^Xp' predict-on
  bindkey '^X^P' predict-off
```

\`Prediction' is a sort of dynamic history completion. With `predict-on` in effect, the line editor will try to retrieve a line back in the history which matches what you type. If it does, it will show the line, extending past the current cursor position. You can then edit the line; characters which do not insert anything mostly behave as normal. If you continue to type, and what you type does not match the line which was found, the line editor will look further back for another line; if no line matches, editing is essentially as normal. Often this is flexible enough that you can leave `predict-on` in effect, but you can return to basic editing with `predict-off`.

Note that, with prediction turned on, deleting characters reverses the direction of the history search, so that you go back to previous lines, like an ordinary incremental search; unfortunately the previous line found could be one you've already half-edited, because they don't disappear from the list until you finally hit \`return' on an edited line to accept it. There's another problem with moving around the line and inserting characters somewhere else: history searching will resume as soon as you try to insert the new characters, which means everything on the right of the cursor is liable to disappear again. So in that case you need to turn prediction off explicitly. A final problem: prediction is bad with multi-line buffers.

If prediction fails with `predict-on` active, completion is automatically tried. The context for this looks like \``:completion:predict::::`'. Various styles are useful at this point: \``list`' could be set to `always`, which will show a possible completion even if there is only one, for example. The style \``cursor`' may have the values \``complete`' to move to the end of the word completed, \``key`' to move past the rightmost occurrence of the character just typed, allowing you just to keep typing, or anything else not to move the cursor which is the default behaviour.

The `incremental-complete-word` function allows you to see a list of possible completions as you type them character by character after the first. The function is quite basic; it is really just an example of using various line editor facilities, and needs some work to make a useful system. It will understand `DEL` to delete the previous character, return to accept, `^G` to abort, `TAB` to complete the word as normal and `^D` to list possibilities; otherwise, keys which do not insert are unlikely to have a useful effect. The completion is done behind the scenes by the standard function `complete-word`.

6.7: Matching control and controlling where things are inserted

The final matter before I delve into the system for writing new completion functions is matching control; the name refers in this case to how the matching between characters already typed on the command line and characters in a trial completion is performed. This can be done in two ways: by setting the `matcher-list` style, which applies to all completions, or by using an argument (`-M`) to the low-level completion functions. Mostly we will be concerned with the first. All this is best illustrated by examples, which are taken from the section `**Matching Control**' in the `zshcompwid` manual page; in the printed manual and the `info' pages this occurs within the section ``Completion Widgets`'.

The `matcher-list` style takes an array value. The values will be tried in order from left to right. For example,

```
  zstyle ':completion:*' matcher-list 'm:{a-z-}={A-Z_}' \
         'r:|[-_./]=* r:|=*'
```

tries the first specification, which is for case-insensitive completion, and if no matches are generated tries the second, which does partial word completion; I'll explain both these specifications in detail as we go along. You can make it do both forms the second time round simply by combining the values with a space, i.e. the last word on the command line becomes `'m:{a-z-}={A-Z_} r:|[-_./]=* r:|=*'`. It is also perfectly valid to have a first matcher empty, i.e. `''`; this means that completion is tried with no matching rule the first time, and will only go on to subsequent matchers in the list if that fails. This is quite a good practice as it avoids surprises.

6.7.1: Case-insensitive matching

To perform case-insensitive matching for all completions, you can set:

```
  zstyle ':completion:*' matcher-list 'm:{a-z}={A-Z}'
```

The ``m:`' specifies standard matching, with the ``{a-z}`' describing what's on the command line, and the ``{A-Z}`' what's in the trial completion. The braces indicate `correspondence classes', which are not lessons taken by email (that's a joke), but a relative of the more usual character classes like ``[a-z]`', which, as you no doubt know, would match any of the letters between `a` and `z`. In this context, with the braces, the letters are forced to match on the left and right hand side of the ``=`', so an ``a`' on the command line must match an ``A`' in the trial completion, a ``b`' must match a ``B`', and so on. Since an `a` in the command line will always match an ``a`' in the trial completion, matcher or no matcher, this means that if you type an ``a`' it will match either ``a`' or ``A`' --- in other words, case-insensitively. The same goes for any other lowercase letter you type. The difference from ``m:[a-z]=[A-Z]`' is that, because ordinary character classes are unordered, *any* lowercase letter would have matched *any* uppercase letter, which isn't what you want. The rest of the shell doesn't know about correspondence classes at all.

Finally, the use of a lowercase ``m`' at the start means that the characters actually inserted onto the line are those from the trial completion --- if you type ``make<TAB>`', the completion process generates file names, and `matcher-list` allows what you type to match the file ``Makefile`', then you need the latter to be inserted on the command line. Use of ``M:`' at the start of the matcher would keep whatever was on the line to begin with there.

If you want completely case-insensitive matching, so that typing ``MAKE<TAB>`' would also potentially complete to ``Makefile`' or ``makefile`' (and so on), the extension is fairly obvious:

```
  zstyle ':completion:*' matcher-list 'm:{a-zA-Z}={A-Za-z}'
```

because now as well as ``a`' matching ``A`', ``A`' will match ``a`' --- and, of course, ``a`' and ``A`' each still match themselves.

More detail on the patterns: they do not, in fact, allow all the possible patterns you can use elsewhere in the shell, since that would be too complicated to implement with little extra use. Apart from character classes and correspondence classes, you can use ``?`' which has its usual meaning of matching one character, or literal characters, which match themselves; or the pattern for the trial completion only can be a single ``*`'. which matches anything. That's it, however; you can't do other things with the ``*`' since it's too difficult for the system to guess what characters should be covered by it.

For the same reason, the `*' must be in an *anchored* pattern, the idea behind which is shown in the next example.

6.7.2: Matching option names

I explained back in chapter 1 that zsh didn't care too much how you specified options: `noglob' and `NOGLOB' and `No_Glob' and `__NO_GLOB_' are all treated the same way. Also, this is the negation of the option `glob'. Having learnt how to match case-insensitively, we have two further challenges: how to ignore a `_' anywhere in the word, and how to ignore the NO at the beginning so that we can complete an unnegated option name after it.

Well, here's how. Since you don't want this for all completions, just for option names, I shall show it as an argument for the `compadd' command, which gives the system the list of possible completions. The option names should then appear as the remaining arguments to the command, and the easiest way of doing that is to have the zsh/parameter module loaded, which it always is for new completion, and use the keys of the special associative array $options:

```
  compadd -M 'B:|[nN][oO]= M:_= M:{A-Z}={a-z}' - ${(k)options}
```

Here, we're interested in the thing in quotes --- it means exactly the same here as it would as an element of the matcher list, except that it only applies to the trial completions given after the `-'. It's in three bits, separated by spaces; as they're in the same word, all are applied one after the other regardless of any previous ones having matched.

Starting from the right, you can see that the last part matches letters case-insensitively; the capital `M' means that, this time, the letters on the command line, not those in the trial completion are kept; this is safe because of the way options are parsed, and reduces unexpected changes.

Moving left, you can now guess `M:_=': it means that the `_' matches nothing at all in the trial completion --- in other words, it is simply ignored. The rule for matching across the `=' is that you move from left to right, pairing off characters or elements of character classes as I already described, and when you run out, you treat any missing characters as, well, missing.

The first part has an `anchor', indicated by what lies between the `:' and the `|'. The B specifies that the case insensitive match of `no' must occur at the start of the word on the command line (with `b' it would be the word in the list of matches), but here it is lax enough to allow this to happen after the `M:_=' has stripped any initial underscores away. Hence it matches no, NO, No or nO at the start of the string, and, just like the `M:_=' part, it ignores it, since there's nothing on the right. Again, the capital `B' at the start means keep what's on the command line: that's important in this case, since if you lost the `no', the meaning would change completely.

So consider the combined effect when trying to complete NO_GL. The first specification allows it to match against _GL; the second allows it to match against GL; the third, against gl; and finally the usual effect of completion means that any option beginning gl may be completed. Try `setopt NO_GL^D' and you should see something like:

```
  NO_GLob            NO_GLobassign     NO_GLobdots
  NO_GLobalrcs       NO_GLobcomplete   NO_GLobsubst
```

--- after the bit you've typed, the form of the words reverts to whatever's in the trial completion, i.e. lowercase letters with no `_'s.

6.7.3: Partial word completion

This example shows the other sort of anchoring, on the right, and also how to use a `*' in the right hand part of a pattern. Consider:

```
  zstyle ':completion:*' matcher-list 'r:|.=* r:|=*'
```

The `` `r:' `` specifies a right-anchored match, using the characters from the trial completion rather than what's already on the command line. As the anchor is on the right this time, the pattern (between `` `:' `` and `` `|' ``) is empty, and its anchor (between `` `|' `` and `` `=' ``) is `` `.' ``. So this specifies that nothing --- a zero length string, or a gap between characters if you want to think of it like that --- when followed by a `` `.' ``, matches anything at all in the trial completion.

Consequently, the second part says that nothing anchored on the right by nothing --- in other words, the right hand end of the command line string --- matches anything. This is what completion normally does, add anything at all at the end of the string; we've added this part to the matcher in case the cursor is in the middle of the word. It means that the right hand end will always be completed, too.

Let's see that in action. Here are the actual contents of my actual `tmp` directory, never mind why:

```
  regframe.rpm  t.c  testpage.dvi  testpage.log  testpage.ps
```

Now I set the `matcher-list` style as above and type:

```
  echo t.p<TAB>
```

and get

```
  echo testpage.ps
```

So, apart from the normal completion at the end (`p` to `ps`), the empty string followed by a `.` was allowed to match anything, too, and I got the effect of completing both bits of the word.

You might wonder what happens when there's a file `testpage.old.ps` around, i.e. the anchor appears twice in that. With the matcher set as given above, that won't be completed; the anchor needs to be matched explicitly, not by a wildcard. If you don't like that, you can change the `` `*' `` after the `` `=' `` in the specification to `` `**' ``; this form allows the anchor to occur in the string being matched. You can think of `` `*' `` and `` `**' `` as taking the shortest and the longest possible matches respectively. If you use a lot of `` `**' `` specifications in your matches, things can get very confusing, however.

Other shells have a facility for completing inside words like this, where it goes by such names as `enhanced' completion, although it is usually not so flexible. In the case of tcsh, not just `` `.' `` but also `` `-' `` and `` `_' `` have this effect. You can force this with

```
  zstyle ':completion:*' matcher-list 'r:|[._-]=* r:|=*'
```

6.7.4: Substring completion

I've mentioned `` `r' `` and `` `B' ``, but corresponding to `` `r' `` there is `` `l' ``, which anchors on the left instead of the right, and corresponding to `` `B' `` there is `` `E' `` which matches at the end instead of the beginning; and, of course, all exist in both upper- and lowercase forms, meaning `keep what the user typed' and `keep what is in the list of possible matches', respectively.

Here is an example of using `` `l:|=*' `` to match anything at the start of the word: this is the effect of having an empty anchor, as you saw with `` `r' `` above, but note with `` `l' ``, the anchor appears, logically enough, on the left of the `` `|' ``, in the order they would appear on the command line. By combining this with the `` `r' `` form, you can make the completion system work when what is on the command line matches only a substring of a trial completion --- i.e., has anything else on the left and on the right. Since this can potentially generate a lot of matches, it might by an idea to try it after any other matcher specifications you have. So the following tries case-insensitive completion, then partial-word completion (case-sensitively), then substring completion:

```
  zstyle ':completion:*' matcher-list 'm:{a-z}={A-Z}' \
         'r:|[._-]=* r:|=*' 'l:|=* r:|=*'
```

6.7.5: Partial words with capitals

This section illustrates another feature: if you use `||' when specifying anchors for `L' or `R' or their lowercase variants, the pattern part for what appears on the command line, which would usually be translated into some other pattern, is treated instead as another anchor on the other side of the pattern --- which isn't matched against the pattern in the word, it just has to appear. In other words, this part matches without being `swallowed up' in the process. An example (again adapted from the manual) will make this clearer.

```
  compadd -M 'r:[^A-Z0-9]||[A-Z0-9]=** r:|=*' \
          LikeTHIS LooHoo foo123 bar234
```

The four possible completions are on the second line. The second of the two matcher specifications just allows anything to match on the right, so if we are inside the word, the remainder may be completed. The first word is where the action is; it says `A part of the completion which has on the left something other than an upper case letter or a digit, and on the right an upper case letter or a digit, may match anything, including the anchor'. So in particular, this would allow `LH' to complete to `LooHoo' --- and only that, since `LikeTHIS' has an uppercase letter to the left of the `H', which is not allowed. In other words, the chunks of word beginning with uppercase letters and digits act like the start of substrings. (If you like, remember that last sentence and the specification, and forget the rest.)

6.7.6: Final notes

To put everything together, the possible specifications are `m:...=...', `l:...|...=...', `r:...|...=...', `b:...|...=...' and `e:...|...=...', which cause the command line to be altered to the match found, and their counterparts with an uppercase letter, which cause what's already on the command line to be left alone and the remaining characters to be inserted directly from the completion found. The `...' are patterns, which all use the same format. They can include literal characters, a `?', and character or correspondence classes, while the rightmost pattern in each type may also consist of a `*' on its own. Characters are matched from left to right; a missing character matches an empty string, `*' matches any number of characters. Specifications may be joined in a single string, in which case all parts will be applied together.

When using the matcher-list style, a list of different specifications can be given; in this case, they will be tried in turn until one of them generates matches, and the rest will not be used.

There's another style apart from matcher-list, called matcher. This can be set for a particular context, possibly with specific tags, and will add the given matcher specifications using exactly the same syntax as matcher-list for that context, except that here all specifications are used at once, even if they are given as different elements of an array. This is possibly useful because matcher-list is only aware of the completer, not of any more specific part of the context.

Although I won't talk about matching control after this section, there may be cases where you want to include `compadd -M ...' in a completion function of your own to help the user. Many of the existing completion functions provide partial word completion where it seems useful; for example, completion of zle functions allows i-c-w to be completed to incremental-complete-word in this way.

Actually, you can configure this to a considerable extent without altering a function, using styles and labelled tags. From the manual:

```
  zstyle ':completion:*:*:foo:*' tag-order '*' '*:-case'
  zstyle ':completion:*-case' matcher 'm:{a-z}={A-Z}'
```

In command foo, whatever the tags are, they are to be tried normally first (the `*' argument to tag-order), then under the same name with `-case' appended. The second style defines a matcher for any tag ending in the suffix `-case', which allows lowercase characters to match uppercase ones. The upshot is that completion of anything at all for the command foo will be tried first case-sensitively, then case-insensitively.

6.8: Tutorial

Before bamboozling you with everything there is to know about writing your own completion function, I'll give you an example of something I wrote myself recently. If you were doing this yourself, you would then just stick this function somewhere in your function search path, and next time you started the shell it would start doing its work. However, the file already exists: it's called `_perforce` and you should find it in the function search for versions 4.1.1 and above of zsh. I apologize if it's not the ideal function to start with, but it is fresh in my mind, so what I'm saying has some chance of being correct.

This section is subtitled, `How I struggled to write a set of completions for Perforce'. Perforce is a commercial configuration management tool (as they now call revision control systems); consult http://www.perforce.com/ for details. It's concepts aren't a million miles from CVS, the archetypal system of this kind, but it was sufficiently different that the completion functions needed rewriting from the ground up. You won't need to know anything about CVS or Perforce, because at each stage I'll explain what I'm trying to complete and why. This should give you plenty of meat for writing completions of your own. After the tutorial, the chapter goes into the individual details, which will expand on some of the things that appeared briefly in the tutorial.

What I tend to find the most complicated part of this is making sure the completion system knows the correct types of completions and their tags to be completed at once. This probably won't be your first priority when trying to write completions of your own, but if you do it right, all the stuff about selecting types and arranging them in groups that I showed above will just work. In this tutorial we arrange to use enough of the higher level functions that it will work without too much (apparent) effort. Of course, working out from scratch which those functions are isn't always that easy; hence the tutorial.

Needless to say, I will simplify grossly at a lot of points. You can see the finished product in the zsh 4.1 distribution. It even has a few comments in.

Basic structure

Like the `cvs` command and a few other of the more complicated commands you might use, Perforce is run by a single command, `p4`, followed by an argument giving the particular Perforce command, followed by an options and arguments to that command.

This dictates the basic tasks the completion functions must do:

- If we are in the first argument, complete the name of the subcommand.
- If we are in a subsequent argument, look up the name of the subcommand and call the function which handles its arguments.

This is more complicated than most commands you will write completions for. However, one useful feature of the completion system is you can do completions in a recursive fashion. So once you get to the point where you are handling arguments for a particular subcommand, you can completely forget about the first step --- as if the subcommand was the command on the line.

In addition to the subcommands, there are lots of other types of object Perforce knows about: files, obviously, plus revisions of files, set of changes (`changelists') applied at once, numbers of fixes applied to files (essentially a way of tying changlists to a particular change request for bugtracking purposes), types of file --- text, binary, etc., and several others. We will break down each of these completions into its own function. That means that any time we need to complete a particular type of object, wherever it appears (and many of these objects can appear in lots of different places), we just call the same function.

Hence there are a large number of different functions:

- The main dispatcher for the command, called `_perforce` for clarity --- the main command it handles is ``p4'`, but the name Perforce is more familiar.

- One function for each subcommand.
- One function for each type of object Perforce knows about and we complete (we don't bother completing dates, for example).
- In some cases, in particular files, multiple functions since there are different types of file --- regular files and directories completed in the normal way, files completed by asking Perforce where it has stored them, files opened for some form of change to be made to them, and so on. Each of these is completed by a different function.

This makes it impractical to put all the functions in separate files since editing them would be a nightmare. What's more, since we will always go through the dispatcher `_perforce`, we don't need to tell the shell to autoload all the other functions; it can just hook them in from the main file. The file `_perforce` therefore has the structure:

```
#compdef p4
# Main dispatcher
_perforce() {
  # ...
}

# Helper functions for the various types of object

_perforce_files() {
  # ...
}

# ...

# Dispatchers for the individual subcommands.

_perforce_cmd_help() {
  # ...
}

# Code to make sure _perforce is run when we load it
_perforce "$@"
```

That last line is probably the least obvious. It's because of the fact that zsh (unlike other shells) usually treats the file of an autoloaded function as being the body of the function. Since everything else here just defines a function, without the last line nothing would happen the first time it was run; it would define `_perforce` and all the other functions, but that was it. The last line makes sure `_perforce` gets run with all the arguments passed down. The shell is smart enough to know that the `_perforce` function we defined in the file is the one to keep for future use, not the entire file, so from then on things are easy; we just have a complete set of ready-defined files.

In fact the various helper functions didn't even need to use the `_' convention for completion functions, since the completion system didn't see them directly. However, I've kept it for consistency.

There's one extra trick: apart from `_perforce` itself, the function definitions look like this:

```
(( $+functions[_perforce_cmd_diff] )) ||
 _perforce_cmd_diff() {
   # body of function
 }
```

This is to allow the user to override each function separately. The test uses the `$functions` special associative array from the `zsh/parameter` module, which the completion system loads. If the function is already defined, because the corresponding element in the `$functions` parameter is set, then we skip the definition of the function here, because the user has already defined it. So if you were to write your own `_perforce_cmd_diff` and put it into the function path, it would be used, as you no doubt intended.

6.8.1: The dispatcher

This top level is only necessary for complex commands with multiple subcommands. There are interesting titbits here, but if you just want to know how to complete a command with ordinary UNIX-style argument parsing, skip to the next section.

The main `_perforce` function has the two purposes described at the top of the previous subsection. We need to decide whether we are in the first word after the `p4` command itself. A simple way of doing that is:

```
  if (( CURRENT > 2 )); then
    # Remember the subcommand name
    local cmd=${words[2]}
    # Set the context for the subcommand.
    curcontext="${curcontext%:*:*}:p4-$cmd"
    # Narrow the range of words we are looking at to exclude `p4'
    (( CURRENT-- ))
    shift words
    # Run the completion for the subcommand
    _perforce_cmd_$cmd
  else
    local hline
    local -a cmdlist
    _call_program help-commands p4 help commands | while read -A hline; do
      (( ${#hline} < 2 )) && continue
      [[ $hline[1] = (#i)perforce ]] && continue
      cmdlist=($cmdlist "${hline[1]}:${hline[2,-1]}")
    done
    _describe -t p4-commands 'Perforce command' cmdlist
  fi
```

This already looks a bit horrific, but it breaks down quite easily. We test the `$CURRENT` parameter, which is a special parameter in the completion system giving the word on the command line we are on. This is the syntactic word --- the completion system has already done the hard job (and that's not an overstatement, I can tell you) of deciding what makes up a word on the command line, taking into account quoting and special characters. The array of words is stored, unsurprisingly, in the array `$words`. So word 1 will be `` `p4' `` and word 2 the subcommand.

Hence if we are past word 2, we look at `${words[2]}` to get the subcommand, and use that to decide what to do next. The change to `$curcontext` is a bit of cleverness to make it easy for the user to defined styles for particular subcommands; refresh your mind by looking at the discussion of styles and contexts above if you need to. For example, if you are completing after `` `p4 diff' ``, the context will look something like `` `:completion::complete:p4-diff:argument-1:opened-files' `` where the remainder says you are on the first argument and are complete the tag `` `opened-files' ``, We'll see down below how we tell the system to use that tag; the `` `argument-1' `` is handled by the `_arguments` utility function, which takes away a lot of the load of handling options and arguments in a standard UNIX format.

Next, we pretend that the `` `p4' `` at the start wasn't there by removing the front of `$words` and decrementing `$CURRENT` so as to reflect its new position in `$words`. The reason for doing this is that we are going to use `_arguments` for handling the subcommand. As is only sensible, this function looks at the first element of `$words` to find the command word, and treats the rest as options or arguments to the command.

We then dispatch the right function for the command simply by constructing the name of the function on the fly. Of course it's a little neater to check the function exists first; `$+functions[_perforce_cmd_$cmd]` would come to our aid again.

However, if we're still on the second (original) word, we have to generate a list of functions to complete. We will do this by asking Perforce's help system to list them, and store the results in the array `$cmdlist`. The loop has a couple of checks to remove blank lines and the title line at the start. The remaining lines have a command and a description. We take the command, but also tack the description on after a colon --- we can then show the user the description, too, as a bit of extra help.

Actually, the Perforce command that generates the list of subcommands is simply `` `p4 help command' ``. (That's really all you need to know; skip the rest of the paragraph if you just want the basics.) The `` `_call_program help-commands' `` was stuck in front for the name of configurability. Before executing the command, the system checks in the current context with the given tag `help-commands` for the style `command`. If it finds a value for that style, it will use that as the command to execute in the place of the remaining arguments. If the style it read began with `-`, then the command it was going to execute --- i.e. `` `p4 help commands' `` is appended to the end of the command read from the style, so that the user's command can process the original command if it needs to. This is really extreme sophistication; you will rarely actually need the `command` style, but if you are writing a completion for others to use it's polite to give them a chance to intercept calls in this way.

The `_describe` command then does the work for us. The `` `-t p4-commands' `` gives the tag we are going to use; the convention is that tag names are plural, though there's nothing to enforce this. Then we give an overall description --- this is what appears after `` `Completing ' `` in the examples of the `format` style above; if you don't have that set, you won't see it. Finally, we give the array name --- note it is the *name*, not the substituted value. This is more efficient because the shell doesn't need to extract the values until the last minute; until then it can pass around just the single word. The `_description` function knows about the `` `completion:description' `` syntax; reread what I said about the `verbose` style for what the system does with the descriptions for the completion.

The `_describe` function is one level above the completion system's basic builtin command, `compadd`; it just knows about a single tag, with a little icing sugar to display verbose descriptions. Later, we'll see ways of building up alternatives where different types of completion can be completed at the same point. There are lots of ways of doing this; some of the more complicated are relegated to the detailed descriptions that follow the tutorial.

6.8.2: Subcommand completion: `_arguments`

Suppose we are now completing after `` `p4 diff' ``. We have altered the command line so that the function now sees the `` `diff' `` as the first word, as if this were the command. This makes the next step easier; the `_arguments` function won't see irrelevant words on the command line, since it is designed to handle the arguments to a simple command in the standard form `` `command [ options ] arguments ...' ``. Here's the simple version.

```
  _perforce_cmd_diff() {
      _arguments -s : \
          '-f[diff every file]' \
          '-t[include non-text files]' \
          '(-sd -se -sr)-sa[opened files, different or missing]' \
          '(-sa -se -sr)-sd[unopened files, missing]' \
          '(-sa -sd -sr)-se[unopened files, different]' \
          '(-sa -sd -se)-sr[opened files, same as depot]' \
          '-d-[select diff option]:diff option:'\
  '((b\:ignore\ blanks c\:context n\:RCS s\:summary'\
  'u\:unified w\:ignore\ all\ whitespace))' \
          "*::file:_perforce_files"
  }
```

I've split the argument beginning `-d` into three lines to fit, but it's just a single argument. Also, for clarity I've missed out the line with the `` `$+functions' `` test to see if `_perforce_cmd_diff` was already defined; I'll forget about that for now.

The function `_arguments` has been described as having `the syntax from hell', but with the arguments already laid out in front of you it doesn't look so bad. The are three types of argument: options to `_arguments` itself, arguments saying how to handle options to the command (i.e. `p4 diff'), and arguments saying how to handle normal arguments to the command.

The first two are for _arguments itself; `-s' tells it that single-letter options are allowed, i.e. they can be combined as in `-ft'. Luckily for our purposes, that doesn't stop us having multiple word options, too. The colon on its own then says everything else is an argument relating to the command line being handled.

We then start off with some simple options; as you can probably guess straight away, the first two say that `p4 diff -f' passes a flag to say any file can be diff'ed (not just ones open for editing), and that `p4 diff -t' passes a flag to say that binary files can be diff'ed (not just text files). Note the use of square brackets for giving a description; this is handled by the verbose style as I mentioned for _describe. In fact, the list of possible options and arguments, suitably rearranged, will end up passing through _describe. The descriptions in square brackets are optional, as the use of square brackets might suggest; you could just have `-f' and `-t' (making it fairly obvious why the `:' to separate off _arguments's own options is a good idea).

The next step in complexity is that set of functions with the list in parentheses in front. These give mutually exclusive options. In other words, if there's already a -sa on the command line, don't complete any of -sd, -se or -sr, and so on. (Remember that by default you need to type the first `-' of an option, or the system will go straight to normal arguments, which we'll come to in a moment.)

Next comes the specification for the option -d. All those colons indicate that this option has an argument, and the - following straight after the -d indicates that it has to be in the same word, i.e. follow the -d without a space. After the first colon comes a description for the argument. This is what you see when you try to complete the after -d; compare this with the expression in square brackets before, which is what you see when you try to complete the -d itself. Then after the second colon is an expression saying how to complete that argument.

This final part of the specification for an option with an argument can take various forms. The simplest is just a single space; this means there's nothing to complete, but the system is aware the user needs to type something for that word and can prompt with the description. The next simplest is a set of words in parentheses: here, we could have had `(b c n s u w)'. Instead, we've had a variant on that which gives yet another set of descriptions, namely those for the individual completions that appear after -d. Note various things: the parentheses are doubled and the colons and spaces within the completion options are backslashed. All of these are simply there to make it easy for _arguments to parse the string. The upshot of this is that in the following context:

```
 p4 diff -d
```

a verbose completion using the format style as described above looks like:

```
  Completing diff option
  b  # ignore blanks
  c  # context
  n  # RCS
  s  # summary
  u  # unified
  w  # ignore all whitespace
```

or similar --- I have the list-separator style set to `#', because it looks like a comment normal shell syntax, but in your case you may get `--' as the separator.

(In case you were wondering why the colons needed to be quoted when it seemed you'd already got to the last argument: it's possible for options to have multiple arguments, and you can continue having sets of :*description*:*action* pairs. This means the system needs some way of distinguishing these colons from ones inside arguments. While I'm digressing, you may also have noticed that I could have written the -s*X* as an option with arguments, in which case you can have a bonus point.)

The final argument starts with a `` `*' ``, which means it applies to all remaining arguments to `` `p4 diff' `` after the options have been processed. Most of the rest is similar to the form for options, except for the doubled colon, which indicates that `$CURRENT` and `$words` should be altered to reflect only the arguments being handled by this argument specifier --- exactly what we did before calling `_perforce_cmd_diff` in the first place, in fact. As we mentioned before, this makes the next step of processing easier if happens to call `_arguments` again. (Actually it doesn't in this case.) The `` `file' `` then describes the arguments and the final part, `_perforce_files`, tells the system to call that function to complete a file name.

There are numerous (it sometimes seems, endless) subtleties to `_arguments`. I won't try to go into them in the tutorial; see the description of `_arguments` below for something more detailed to refer to, and if you are feeling *really* brave look at the description in the `zshcompsys` manual page. Even better, dig into one of the existing completion functions --- something handling completion for a UNIX command is probably good, since these make heavy use of `_arguments` --- and see how those work. Despite the complexities, I would definitely suggest using `_arguments` wherever possible to take away any need on your part to do processing of command line arguments.

6.8.3: Completing particular argument types

Now we'll look inside the `_perforce_files` function as an example of the nitty gritty of completing one particular type of argument, which might have some quite complicated internal structure. This is true in Perforce as the filename can have extra information tacked on the end: `` `file#``*revision*`'` indicates the revision of a file, `` `file@``*change*`'` indicates a change status, and in some cases you can get `` `file@``*change1*`,`*change2*`'` to indicate a range of changes (likewise revisions). Furthermore, `file` can be specified in different ways, and the file to be completed may be limited by some kind of context information. We'll start from simple filenames and gradually add these possibilities in.

Different types of file, part 1

There are so many possibilities for files that I'm going to split up `_perforce_files` into individual functions handling different aspects. For example, even if we are just handling ordinary files in the way the completion system normally does, Perforce commands understand a special file name `` `...' `` which means `every subdirectory to any depth'. (Interestingly, zsh used to have this to mean the same thing, instead of `` `**' ``; it was changed in zsh because as the `` `.' ``'s are regular characters there's no easy way of quoting them. You didn't need to know this.)

I'm going to say we can complete both like this:

```
  _alternative \
      "files:file:_path_files" \
      "subdirs:subdirectory search:_perforce_subdir_search"
```

The function `_alternative` is a little bit like `_arguments`, but thankfully much simpler. It's name gives away its purpose; every argument specifies one of a set of possible alternatives, all of which are valid at that point --- so the user is offered anything which matches out of the choices, unlike `_arguments`, which has to decide between the various possibilities. It's a sort of glorified loop around `` `_describe' ``, with `_arguments`'s conventions on the action for generating completions (up to a point --- `_alternative` doesn't have all the whackier ones, though it does have the ones I've been talking about so far).

Each set of possibilities consists of the name of a tag, a description, and an argument. The tag isn't present in `_arguments`. If you use `^xh` to tell you about valid tags, you'll see `_arguments` has its own generic tag, `argument-rest`; this isn't usually all that useful, so we are going to supply more specific ones.

In the first possibility, it's the standard one for files, `` `files ``. The function is the basic low-level one for completing files, too; it's described below, but you already know a lot about the effect since it's the completion system's workhorse which you use it all the time without realising. Actually, it will supply its own tags, but that doesn't matter since they will silently override what we say.

The second possibility is the new one we're adding. I've therefore invented a suitable tag `subdirs', a description, `subdirectory search', and the name of the function I'm going to supply to do the completion. This is quite simple:

```
 _perforce_subdir_search() {
    compset -P '*/'
    compadd "$@" '...'
  }
```

The first line tells the completion system to ignore anything up to the last `/'. That's so we can append a `...' to any directory which already exists on the command line. The builtin compset does various low-level transformations of this time. Note that the -P is `greedy' --- it looks for the longest possible pattern match, which is the usual default in zsh and other UNIX pattern matchers.

The second line actually adds the `...' as a completion; compadd is the key builtin for the whole completion system. I've actually passed some on the arguments which we got to `_perforce_subdir_search' via `"$@"'. In fact, looking back it seems as if there weren't any! However, _alternative actually passed some behind my back --- and it's a good thing, too, since it's exactly those arguments that give the tag `subdirs' and the description `subdirectory search'. So that extra `"$@"' is actually quite important. The buck stops here; there's nothing below compadd. A function of this simplest only works well when the handling of tags and contexts has already been done; but we just saw that _alternative did that, so as long as we always call _perforce_subdir_search suitably, we're in the clear.

Different types of file, part 2

Furthermore, a Perforce file specification can look like a normal UNIX file path, or it can look like:

```
  //depot/dirs/moredirs/file
```

(don't get confused with paths to network resources, which also use the doubled slash or backslash on some systems, notably Cygwin). We could use _alternative to handle this, too, and if I was writing _perforce again I probably would for simplicity. However, I decided to do it just by testing for the `//' in _perforce_files. This means that the structure of p4_files so far looks like:

```
  if [[ $PREFIX = //* ]]; then
    # ask Perforce for files that match
    local -a altfiles
    altfiles=(
      'depot-files:file in depot:_perforce_depot_files'
      depot-dirs:directory in depot:_perforce_depot_dirs'
    )
    # add other alternatives, such as the `...' thing
    altfiles=($altfiles
      "subdirs:subdirectory search:_perforce_subdir_search"
    )
    _alternative $altfiles
  else
    _alternative \
        "files:file:_path_files" \
        "subdirs:subdirectory search:_perforce_subdir_search"
  fi
```

where we are still to write the functions for the first two alternatives in the first branch; the `...' is still valid for that branch, so I've added that as the third alternative. I've used the array $altfiles because, actually, the structure is more complicated than I've shown; doing it this way makes it easier to add different sets of alternatives.

The choice of which branch is made by examining the $PREFIX special variable, which contains everything (well, everything interesting) that comes before the cursor position in the word being completed. There is a counterpart $SUFFIX which we will see in a moment. The `almost everything' comes because sometimes we definitely don't want to see the whole $PREFIX. Completing the three dots was such as case --- we didn't want to see anything up to the last /. What that `compset -P '*/'' actually did was move the matched pattern from the front of $PREFIX to the end of $IPREFIX, another special parameter which contains parts of the completion we aren't currently interested in, but which are still there. This allows us to concentrate on a particular part of the completion. However you do that --- whether by compset or directly manipulating $PREFIX and friends --- the completion system usually restores the parameters when you exit the function where you altered them. This fits in nicely with what we're doing here with _alternative --- if we handle adding `...' by ignoring everything up to the last slash, for example, we don't want the next completion we try to continue to ignore that; other file completions will want to look at the directory path.

`Depot' is Perforce's name for what CVS calls a repository --- the central location where all versions of all files are stored, and from where they are retrieved when you ask to look at one. I've separated out `depot-dirs' and `depot-files' for various reasons. First, the commands to examine files and directories are different, so the completion function is different. Second, we can offer different tags for files and directories --- this is what _path_files does for normal UNIX files. Third, it will later allow us more control --- some commands only operate on directories. Here's _perforce_depot_files; _perforce_depot_dirs is extremely similar:

```
  _perforce_depot_files() {
    # Normal completion of files in depots
    local pfx=${(Q)PREFIX} expl
    local -a files

    compset -P '*/'
    files=(${${${(f)"$(\
_call_program files p4 files \
\"\$pfx\*\$\{\(Q\)SUFFIX\}\" 2>/dev/null)"}%\#*}##*/})
    [[ $#files -eq 1 && $files[1] = '' ]] && files=()
    compadd "$@" -a files
}
```

A little messy (and still not quite the full horror). I've split the key line in the middle which fetches the list from Perforce to make it fit. If you ploughed through chapter 5, you'll recognised what's going on here --- we're reading a list of files, one per line, from the command `p4 files', and we're stripping off the directory at the front, and everything from a `#' on at the end. The latter is a revision number; we're not handling those at this point, though we will later.

Notice the way I remembered $PREFIX before I told the system to ignore it for the word we're now completing. I remembered it as `${(Q)PREFIX}' in order to remove any quotes from the name. For example, if the name on the line so far had a space, $PREFIX (which comes from what is on the command line without any quotes being stripped) would have the space quoted somehow, e.g. `name\ with\ space'. We arrange for $pfx to contain `name with space', which is how Perforce knows the file, using the (Q) parameter flag. We then pass the argument "$pfx*${(Q)SUFFIX}" to `p4 files'; this generates matching files internally. The extra layer of backslash-quoting is for the benefit of _call_program, which re-evaluates its arguments; this ensures the argument is expanded at the point it gets passed to p4 files. All this goes to show just how difficult getting the quoting right can be.

Once we've got the list of bare filenames, we check to see if the list is just one element with no length. That's an artefact of the the "$(cmd)" syntax; if the output is empty, because its quoted you still get one zero-length string output, which we don't want.

Finally, we pass the result to compadd as before. Again, tags and the description have already been handled and we just need to make sure the appropriate options get passed in with "$@". This time we use the `-a' option which tells compadd that any arguments are array name, not a list of completions. This is more efficient;

compadd only needs to expand the array internally instead of the shell passing a potentially huge list to the builtin.

Handling extra bits on a completion

`Extra bits' on a completion could be anything; common examples include an extra value for a comma-separated list (the `_values` functions is for this), or some kind of modifier applied to the completion you have already. We've already seen an example, in fact, since the principle of handling the directory and basename parts of a file is very similar. The phrase `extra bits' may already alert you to the fact that we are heading towards the deeper recesses of completion.

Anyway, here's how we tack a revision or change number onto the end of a file.

I'll stick with revisions: `*filename#revision*', where *revision* is a number. For the full sophistication, there are three steps to this. First, make it easy for the user to add ``#`' to an existing filename; second, recognise that a ``#`' is already there so that revisions need to be completed; third, find out the actual revisions which can be completed. As a revision is just a number, you might think completing it was a bit pointless. However, given the sophistication of zsh's completion system there's actually one very good reason --- we can supply a description with the revisions, so that the user is given information about the revisions and can pick the right one without running some external command to find out. There was the same sort of rationale behind the ``-d`' option to `p4 diff`; there was just one letter to type, but zsh was able to generate extra information to describe the possibilities, so it wasn't just laziness.

First part: make it easy for the user to add the ``#`'. This actually depends on a new feature in version 4.1 of zsh; in 4.0 you couldn't play the trick we need or grabbing the keyboard input after a completion was finished unless you specified a particular suffix to add to the completion (such as the ``/`' after a directory --- this is historically where this feature came from).

The method is to add an extra argument everywhere we complete a file name. For example, change the `compadd` in `_perforce_depot_files` to:

```
 compadd "$@" -R _perforce_file_suffix -a files
```

where the option argument specifies a function:

```
  _perforce_file_suffix() {
    [[ $1 = 1 ]] || return

    if [[ $LBUFFER[-1] = ' ' ]]; then
        if [[ $KEYS = '#' ]]; then
            # Suffix removal with an added backslash
            LBUFFER="$LBUFFER[1,-2]\\"
        elif [[ $KEYS = (*[^[:print:]]*|[[:blank:]\;\&\|@]) ]]; then
            # Normal suffix removal
            LBUFFER="$LBUFFER[1,-2]"
        fi
    fi
}
```

This has been simplified, too; I've ignored revision ranges in the form *file#rev1*,*rev2*. However, I've handled changes (``@`' following a filename) as well as revisions. You'll see this function looks much more like a zle widget rather than a completion widget --- which is exactly what it is; it's not called as part of the completion system at all. After the specified completion, zle reads in the next keystroke, which is stored in `$KEYS`, and calls this function as a zle widget. This means it can manipulate the line buffer; we only need to look at what is at the left of the cursor, stored in `$LBUFFER`.

The function is called with the length of the suffix added to the function. In this case, it's just a space --- we've finished a normal completion, so the system has automatically added a space to what's on the command line. We therefore check we've just got one single character in the suffix, to avoid getting confused.

Next, we look at what's immediately left of the cursor, which is the last character in `$LBUFFER`, i.e. `$LBUFFER[-1]`, to make sure this is a space.

If everything looks OK, we consider the keys typed and decide whether to modify the line. You may already have noticed that in some cases zsh automatically removes that space by itself; for example, if you hit return --- or any other non-printing character --- or if it's a character that terminates a command such as ``&`' or ``;`'. We emulate that behaviour --- most of the second test is simply to do that. The only differences from normal are if the key typed was ``@`' or ``#`'.

The ``@`' is simple --- we just remove the last character, the same as we do for the other characters. For ``#`', however, we also add a backslash to the command line before the ``#`'. That's because ``#`' is a special character with extended globbing, and the completion system generally runs with extended globbing switched on. Adding the backslash means the user doesn't have to; it's never harmful.

To show the next effect, suppose we complete a file name:

```
 p4 diff fil<TAB>
```

to get:

```
 p4 diff filename _
```

where ``_`' shows the cursor position, and then typed ``#`'; we would get:

```
 p4 diff filename\#
```

with the cursor right at the end.

So far so good. For the second step, we need to modify `_perforce_files` to spot that there is a ``#`' on the line before the cursor, and to call the revision code. To do this we add an extra branch at the start of the ``if`' in `_perforce_files` --- at the start, because any ``#`' before the cursor forces us to look at revisions, so this takes precedence over the other choices. When this is added, the code will look like:

```
  if [[ -prefix *\# ]]; then
    _perforce_revisions
  elif [[ $PREFIX = //* ]]; then
    # as before.
```

In fact, that `-prefix` test is just a fancy way of saying the same thing as the ``[[$PREFIX = *\#]]`' and if I wasn't so hopelessly inconsistent I would have written both tests the same.

So now the third step: write `_perforce_revisions` to complete revisions numbers with the all-important descriptions.

```
 _perforce_revisions() {
    local rline match mbegin mend pfx
    local -a rl

    pfx=${${(Q)PREFIX}%%\#*}
    compset -P '*\#'

    # Numerical revision numbers, possibly with text.
    if [[ -z $PREFIX || $PREFIX = <-> ]]; then
        # always allowed (same as none)
        rl=($rl 0)
        _call_program filelog p4 filelog \$pfx 2>/dev/null |
           while read rline; do
            if [[ $rline = (#b)'... #'(<->)*\'(*)\' ]]; then
```

```
                rl=($l "${match[1]}:${match[2]}")
            fi
        done
    fi
    # Non-numerical (special) revision names.
    if [[ -z $PREFIX || $PREFIX != <-> ]]; then
        rl=($rl 'head:head revision' 'none:empty revision'
                'have:current synced revision')
    fi
    _describe -t revisions 'revision' rl
  }
```

Thankfully, a lot of the structure of this is already familiar. We extract the existing prefix before the `` `#' ``, being careful about quoting --- this is the filename for which we want a list of revisions. We ignore everything in the command argument before the `` `#' ``. After generating the completions, we use the `_describe` function to add them with the tag `` `revisions' `` and the description `` `revision' ``.

The main new part is the loop over output from `` `p4 filelog' ``, which is the Perforce command that tells us about the revisions of a file. We extract the revision number and the comment from the line using backreferences (see previous chapter) and weld them together with a colon so that `_describe` will be able to separate the completion from its description. Then we add a few special non-numerical revisions which Perforce allows, and pass this list down to `_describe`. The extra `if`'s are a very minor optimization to check if we are completing a numerical or non-numerical revision.

6.8.4: The rest

It's obvious that this tutorial could expand in any number of directions, but as it's really just to point out some possibilities and directions, that would would miss the point. So the rest of this chapter takes the completion system apart and looks at the individual components. It should at least now be a bit more obvious where each component fits.

6.9: Writing new completion functions and widgets

Now down to the nitty gritty. When I first talked about new completion, I explained that the functions beginning `` `_' `` were the core of the system. For the remainder of the chapter, I'll explain what goes in them in more detail than I did in the tutorial. However, I'll try to do it in such a way that you don't need to know every single detail. The trade off is that if you just use the simplest way of writing functions, many of the mechanisms I told you about above, particularly those involving styles and tags, won't work. For example, much of the code that helps with smart formatting of completion listings is buried in the function `` `_description' ``; if you don't know how to call that --- which is often done indirectly --- then your own completions won't appear in the same format as the pre-defined ones.

The easiest way of getting round that is to take a dual approach: read the following as far as you need, but also try to find the existing completion that comes nearest to meeting your needs, then copy that and change it. For example, here's a function that completes files ending in `.gz` (the supplied function which does this has now changed), which are files compressed by the `gzip` program, for use by the corresponding program that does decompression, `gunzip` --- hence the file and function are called `_gunzip`:

```
  #compdef gunzip zcat

  local expl

  _description files expl 'compressed file'
  _files "$expl[@]" -g '*.[gG][zZ]'
```

You can probably see straight away that if you want to design your own completion function for a command which takes, say, files ending in `.exe`, you need to change three things: the line at the top, which gives the names of programmes whose arguments are to be completed here, the description `compressed file' to some appropriate string, and the argument following the -g to something like '*.exe' --- any globbing pattern should work, just remember to quote it, since it shouldn't be expanded until the inside of the function _files. Once you've installed that somewhere in your $fpath and restarted the shell, everything should work, probably following a longer pause than usual as the completion system has to rescan every completion function when it finds there is a new one.

What you might miss is that the first argument to _description, `files', is the all-important mystical tag for the type of completion. In this case, you would probably want to keep it. Indeed, the _files function is used for all file completions of any type, and knows all about the other tags --- globbed-files, directories, all-files --- so virtually all your work's done for you here.

If you're adding your own functions, you will need your own functions directory. This was described earlier in this guide, but just to remind you: all you need to do is create a directory and add it to $fpath in either .zshenv (which a lot of people use) or .zshrc (which some sticklers insist on, since it doesn't affect non-interactive shells):

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

It's best to put it before the standard completion directories, since then you can override a standard completion function simply by copying it into your own directory; that copy will then be found first and used. This is a perfectly reasonable thing to do with any completion function --- although if you find you need to tweak one of the larger standard functions, that's probably better done with styles, and you should suggest this to us.

6.9.1: Loading completion functions: compdef

The first thing to understand is that top line of _gunzip. The `#compdef' tag is what tells the system when it checks through all files beginning with `_' that this is a function implementing a completion. Files which don't directly implement completions, but are needed by the system, instead have the single word `#autoload' at that point. All files are only loaded when needed, using the usual autoloading system, to keep memory usage down.

You can supply various options to the `#compdef' tag; these are listed in the `Initialization' section of the zshcompsys(1) manual page or `**Completion System**' info node. The most useful are -k and -K, which allow you to define a completion command and binding rather than a function used in a particular context. There are also -p and -P which tell the system that what follows is a pattern rather than a literal command name; any command matching the pattern will use that completion function, unless you used -P and a normal (non-pattern) completion function for the name was found first.

For normal #compdef entries, however, what comes next is a list of command names --- or rather a list of contexts, since the form `-context-' can be used here. For example, the function _default has the line `#compdef -default-'. You can give as many words as you like and that completion will be used for each. Note that contexts in the colon-separated form can't appear here, just command names or the special contexts named with hyphens.

The system does its work by using a function compdef; it gets as arguments more or less what you see, except that the function name is passed as the first argument. Thus the _gunzip completion is loaded by `compdef _gunzip gunzip zcat', _default by `compdef _default -default-', and so on. This simply records the name of the function handling the context in the $_comps associative array which you've already met. You can make extra commands/contexts be handled by an existing completion function in this way, too; this is generally more convenient than copying and modifying the function. Just add `compdef <_function> <command-to-handle>' to .zshrc after the call to compinit.

It's also high time I mentioned an easy way of using the completion already defined for an existing function: `compdef newcmd=oldcmd' tells the completion system that the completion arguments for `newcmd' are to be the same as the ones already defined for `oldcmd'; it will complain if nothing is known about completing for oldcmd.

This works recursively; you can now define completions in terms of that for newcmd. If you happen to know the name of the completion function called, you can use that; the following three lines are broadly equivalent:

```
  compdef $_comps[typeset] foo
  compdef _vars_eq foo
  compdef foo=typeset
```

since the completion for typeset is stored in $_comps along with all the others, and this happens to resolve to _vars_eq; but the last example is easier and safer and the intention more obvious. The manual refers to typeset here as a `service' for foo (guess what the shell stores in the associative array element $_services[foo]).

There's actually more to services: when a function is called, the parameter $service is set. Usually this will just be the name of the command being completed for, or one of the special contexts like `-math-'. However, in a case like the last compdef in the list above, the service will be typeset even though the command name may be `foo'.

This is also used in `#compdef' lines. The top of `_gzip' contains:

```
  #compdef gzip gunzip gzcat=gunzip
```

which says that the file provides two services, for gzip and gunzip, and also handles completion for gzcat, but with the service name gunzip. Only a few of the completion functions actually care what service they provide (you can check, obviously, by looking to see if they refer to $service); but you may have uses for this. Note that if you define services with a compdef command, *all* the arguments must be in the *foo=bar* form; the mixed form is only useful after a #compdef inside completion functions.

6.9.2: Adding a set of completions: compadd

Once you know how to make a new completion function, there is only one other basic command you need to know before you can create your own completions yourself. This is the builtin compadd. It is at the heart of the completions system; all its arguments, after the options, are taken as possible completions. This is the list from which the system selects the possibilities that match what you have already typed. Here's a very basic example which you can type or paste at the command line:

```
  _foo() { compadd Yan Tan Tethera; }
  compdef _foo foo
```

Now type `foo ' and experiment with completions after it. If only it were all that simple.

There are a whole list of options to compadd, and you will have to look in the zshcompwid(1) manual page or the `**Completion Widgets**' info node for all of them. I've already mentioned -M and (long ago) -f. Here are other interesting ones. -X <description> provides a description --- this is used by the format style to pass descriptions, and if you use the normal tags system you shouldn't pass it directly; I'll explain this later.

-P <prefix> and -S <suffix> allow you to specify bits which are not treated as part of the completion, but appear on the line none the less. In fact, they do two different things: if the prefix or suffix is already there, it is ignored, and if it isn't, it is inserted. There are also corresponding hidden and ignored prefixes, necessary for the full power of the completion system, but you will need to read the manual for the full story. The -q option is useful with -S; it enables auto-remove behaviour for the suffix you gave, just like / with the AUTO_REMOVE_SLASH option when completing filenames.

-J <group> is the way group names are specified, used by the group-name tag; there is also -V <group>, but the group here is not sorted (and is distinct from any group of the same name passed to -J). -Q tells the completion code not to quote the words --- this is useful where you need to have unquoted metacharacters in the final completion. It is also useful when you are completion something where the result isn't going to be expanded by the shell.

-U tells compadd to use the list of completions even if they don't match what's on the command line; you will need this if your completion function modifies the prefix or suffix so that they no longer fit what's already there. If you use this, you might consider turning on menu completion (using compstate[insert]=menu), since it might otherwise be difficult to select the appropriate completion.

Finally, note the -F and -W options which I describe below for _files actually are options to compadd too.

6.9.3: Functions for generating filenames, etc.

However, for most types of completion the possibilities will not be a simple list of things you already know, so that you need to have some way of generating the required values. In this section, I will describe some of the existing functions you can call to do the hard work. In the next section I will show how to retrieve information from some special parameters made available by the zsh/parameter module.

Files etc.: the function _files

You have already seen _files in action. Calling this with no arguments simply adds all possible files as completions, taking account of the word on the command line to establish directories and so on.

For more specific use, you can give it various options: `-/' means complete directories, and, as you saw, `-g "<pattern>"' gives a filename generation pattern to produce matching files.

A couple of other options, which can be combined with the ones above, are worthy of mention. If you use `-W <dir>', then completion takes place under directory <dir> rather than in the current directory --- it has no effect if you are using an absolute path. Here, `<dir>' can also be a set of directories separated by spaces or, most usefully since it avoids any problems with quoting, the name of an array variable which contains the list of possible directories. This is essentially how completion for cd with the $cdpath array works. So if you have a program that looks for files with the suffix `.mph', first in the current directory, then in a standard directory, say, /usr/local/oomph', you can do this:

```
   local oomph_dirs
   oomph_dirs=(. /usr/local/oomph)
  _files -W oomph_dirs -g '*.mph'
```

--- note there is no `$' before the variable $oomph_dirs here, since it should only be expanded deep inside _files.

The system that implements $fignore and the ignored-patterns style can be intercepted, if you need to, with the option `-F "<pat>"'; `<pat>' is an array of patterns to ignore, in the usual completion format, in other words the name of a real shell array, or a list of values inside parentheses. If you make sure all the tags stuff is handled properly, ignored-patterns will work automatically, however, and in addition extended globbing allows you to specify patterns with exclusion directly, so you probably won't use this feature directly unless you're in one of your superhero moods.

In addition, _files also takes many of the standard completion options which apply to compadd, for convenience.

Actually, the function _path_files is the real engine room of the system. The advantage of using _files is that it prepares all the tags for you, deciding whether you want directories to be completed as well as the globbed files, and so on. If you have particularly specific needs you can use _path_files directly, but you won't get the automatic fallback one directories and all-files. Because it doesn't handle the tags, _path_files is too lowly to do the usual tricks with label loops, i.e. pretending `dog:-setter' is a tag `dog-setter' with the usual completions for `dog'; likewise, it doesn't implement the file-patterns style. So you need to know what you're doing when you use it directly.

Parameters and options

These can be completed by calls to the `_parameters` and `_options` functions, respectively. Both set up their own tags, and `_options` uses the matching control mechanism described above to allow options to be given in all the available forms. As with `_files`, they will also pass standard `compadd` options down to that function. Furthermore, they are all at a high enough level to handle tags with labels: to translate that into English, you can use them directly without any of the preprocessing described later on which are necessary to make sure the styles dealing with tags are respected.

For more detailed control with options, the functions `_set_options` and `_unset_options` behave like `_options`, but the possible completions are limited to options which are set or unset, respectively. However, it's not that simple: the completion system itself alters the options, and you need to enable some code near the top of `_main_complete` (it's clearly marked) to remember the options which were set or unset when completion started. A straw poll based on a sample of two zsh developers revealed that in any case many people don't like the completion system to second guess the options they want to set or unset in this way, so it's probably better just to stick to `_options`.

Miscellaneous

There are also many other completion functions adding matches of a certain type. These can be used in the same way as `_parameters` and `_options`; in other words they do all the work needed for tags themselves and can be given options for `compadd` as arguments. Normally, these functions are named directly after the type of matches they generate, like `_users`, `_groups`, `_hosts`, `_pids`, `_jobs`, etc.

6.9.4: The `zsh/parameter` module

The new completion system automatically makes the `zsh/parameter` module available for use. This provides an easy way of generating arguments for `compadd`. To get the maximum use out of this, you should be familiar with zsh's rather self-willed syntax for extracting bits out of associative arrays. Note in particular `${(k)assoc}`, which expands to a list of the keys of the associative array `$assoc`, `${(v)assoc}`, which expands to just its values (actually, so does `$assoc` on its own), and `${(kv)assoc}` which produces key/value pairs. For all intents and purposes, the keys and values, or the pairs of them, are in a random order, but as the completion system does it's own sorting that shouldn't be a problem. Mostly, the important parts for completion are in the keys, i.e. to add all aliases as possible completions, you need \``compadd ${(k)aliases}`'.

Here's a list of associative and ordinary arrays provided; for more information on the values of the associative arrays, which could be useful in some cases, consult the section **The zsh/parameter Module** in the `zshmodules(1)` manual page or the corresponding info node. First, the associative arrays.

`$aliases`, `$dis_aliases`, `$galiases`
: The keys of these arrays give ordinary aliases, disabled ordinary aliases for those where you have done `disable -a <alias>` to turn them off temporarily, and global aliases as defined with `alias -g`.

`$builtins`, `$dis_builtins`
: The keys give active and disabled shell builtin commands.

`$commands`
: The keys are all external commands stored in the shells internal tables; it does this both for the purposes of fast completion, and to avoid having to search each time a command is executed. It's possible that a command is missing or incorrectly stored if the contents of your `$path` directories has changed since the shell last updated its tables; the `rehash` command fixes it.

`$functions`, `$dis_functions`
: The keys are active and disabled shell functions.

`$history`

Here, the *values* are complete lines stored in the internal history. The keys are the numbers of the history line; it's an associative, rather than an ordinary, array because they don't necessarily start at line 1. However, see the `historywords` ordinary array below.

`$jobtexts`, `$jobdirs`, `$jobstates`
These give you information about jobs; the keys are the job numbers, as presented by the `jobs` command, and the values give you the other information from jobs: `$jobtexts` tells you what the job is executing, `$jobdirs` its working directory, and `$jobstates` its state, where the bit before the colon is the most useful as it refers to the whole job. The remainder describes the state of individual processes in the job.

`$modules`
The keys give the names of modules which are currently available to the shell, i.e. loaded or to be autoloaded, essentially the same principle as with functions.

`$nameddirs`
If you have named directories, either explicitly (e.g. assigning \``foo=/mydir`' and using \``~foo`') or via the `AUTO_NAME_DIRS` option, the keys of this associative array give the names and the values the expanded directories.

`$options`, `$parameters`
The keys give shell options and parameters, and are used by the functions `_options` and `_parameters` for completion, so you will mostly not need to refer to them directly.

`$userdirs`
The keys give all the users on the system. The values give the corresponding home directory, so \``${userdirs[juser]}`' is equivalent to having `~juser` expanded and is thus not all that interesting, except that by doing it this way you can test whether the expansion exists without causing an error.

Now here are the ordinary arrays, which you would therefore refer to simply as `${reswords}` etc.

`$dirstack`
This contains your directory stack, what you see with \``dirs -v`'. Note, however that the current directory, which appears as number 0 with that command, doesn't appear in `dirstack`. Of course it's easy to add it to a completion if you want.

`$funcstack`
This is the call stack of functions, i.e. all the functions which are active at the time the array was referenced. `^Xh` uses this to display which functions have been called for completion.

`$historywords`
Unlike `$history`, this contains just the individual words of the shell's command line history, and is therefore likely to be more useful for completion purposes.

`$reswords`, `$dis_reswords`
The active and disabled reserved words (effectively syntactically special commands) understood by the shell.

Other ways of getting at information

Since the arguments to `compadd` undergo all the usual shell expansions, it's easy to get words from other sources for completion, and you can look in the existing completion functions for many examples. A good understanding of zsh's parameter and command expansion mechanisms and a strong stomach will be useful here.

For example, here is the expansion used by the `_limits` function to retrieve the names of resource limits from the `limit` command itself:

```
  print ${${(f)"$(limit)"}%% *}
```

which you can test does the right thing. Here's a translation: "$(limit)" calls the command in a quoted context, which means you get the output as if it were a single file (just type `limit' to see what that is). ${(f)...} splits this into an array (it is now outside quotes, so splitting will generate an array) with one element per line. Finally, ${...%% *} removes the trailing end of each array element from the first piece of whitespace on, so that `cputime unlimited' is reduced to `cputime', and so on. Type `limit ^D', and you will see the practical upshot of this.

That's by no means the most complicated example. The nested expansion facility is used throughout the completion functions, which adds to brevity but subtracts considerably from readability. It will repay further study, however.

6.9.5: Special completion parameters and compset

Up to now, I've assumed that at the start of your completion function you already know what to complete. In more complicated cases that won't be the case: different things may need completing in different arguments of a command, or even some part of a word may need to be handled differently from another part, or you need to look for a word following a particular option. I will first describe some of the lower level facilities which allow you to manipulate this; see the manual page zshcompwid(1) or the info node **Completion Widgets** for the details of these. Later, I will show how you can actually skip a lot of this for ordinary commands with options and arguments by using such functions as _arguments, where you simply specify what arguments and options the function takes and what sort of completion they need.

The heart of this is the special parameters made available in completion for testing what has already been typed. It doesn't matter if there are parameters of that name outside the completion system; they will be safely hidden, the special values used, and the original values restored when completion is over.

$words is an array corresponding to the words on the command line --- where by a `word' I mean as always a single argument to the command, which may include quoted whitespace. $CURRENT is the index into that array of the current word. Note that to avoid confusion the ksh-like array behaviour is explicitly turned off in _main_complete, so the command itself is $words[1], and so on.

The word being completed is treated specially. The reason is that you may only want to complete some of it. An obvious example is a file with a path: if you are completing at `foo/bar', you don't want to have to check the entire file system; you want the directory foo to be fixed, and completion just for files in that. There are actually two parts to this. First, when completion is entered, $PREFIX and $SUFFIX give you the part of the current word before the cursor, and the remainder, respectively. It's done like this to make it possible to write functions for completing inside a word, not just at the end. The simplest possible way of completing a file is then to find everything that matches $PREFIX*$SUFFIX.

But there's more to it than that: you need to separate off the directory, hence the second part. The parameters $IPREFIX and $ISUFFIX contain a part of the string which will be ignored for completion. It's up to you to decide what that is, then to move the bit you want to be ignored from $PREFIX to $IPREFIX (that's the usual case) or from $SUFFIX to $ISUFFIX, making sure that the word so far typed is still given by $IPREFIX$PREFIX$SUFFIX$ISUFFIX. Thus in completing foo/bar, you would strip foo/ from the start of $PREFIX and tack it onto the end of $IPREFIX --- after recording the fact that you need to move to directory foo, of course. Then you generate files in foo, and the completion system will happily accept barrack or barbarous as completions because it doesn't care about the foo any more.

Actually, this is already done by the the _files and _path_files functions for filename completion. Also, you can get some help using the compset builtin command. In this case, the incantation is

```
  if compset -P "*/"; then
    # do whatever you need to with the leading
    # string up to / stripped off
  else
```

```
    # no prefix stripped, do whatever's necessary in this case
  fi
```

In other words, any initial match of the pattern `*/' in $PREFIX is removed and transferred to the end of $IPREFIX; the command status tells you whether this was done. Note that it is the longest possible such match, so if there were multiple slashes, all will be moved into $IPREFIX. You can control this by putting a number <N> between the -P and the pattern, which says to move only up to the <N>th such match; here, that would be a pattern with exactly <N> slashes. Note that -P stands for prefix, not pattern; there is a corresponding -S option for the suffix. See the manual for other uses of compset; these are probably the most frequent.

If you want to make the test made by compset, but without the side effect of changing the prefixes and suffixes, there are tests like this:

```
  if [[ -prefix */ ]]; then
    # same as with `compset -P "*/"', except prefixes were left alone.
  fi
```

These have the advantage of looking like all the standard tests understood by the shell.

There are three other parameters special to completion. The $QIPREFIX and $QISUFFIX are a special prefix and suffix used when you are dividing up a quoted word --- for example, in `zsh -c "echo hi"', the word "echo hi" is going to be used as a command line in its own right, so if you want to do completion there, you need to have it split up. You can use `compset -q' to split a word in this fashion.

There is also an associative array $compstate, which allows you to inspect and change the state of many internal aspects of completion, such as use of menus, context, number of matches, and so on. Again, consult the manual for more detail. Many of the standard styles work by altering elements of $compstate.

Finally, in addition to the parameters special to completion, you can examine (but not alter) any of the parameters which appear in all editing widgets: $BUFFER, the contents of the current editing line; $LBUFFER, the part of that before the cursor; $RBUFFER, the rest; $CURSOR, the index of the cursor into $BUFFER (with the first character at zero, in this case --- or you can think of the zero as being the point before the first character, which is where insertion would take place with the cursor on the first character); $WIDGET and $LASTWIDGET, the names of the current and last editing or completion widget; $KEYS, the keys typed to invoke the current widget; $NUMERIC, any numeric prefix given, unset if there is none, and a few other probably less useful values. These are described in the zshzle(1) manual page and the **Zsh Line Editor** info node. In particular, I already mentioned $NUMERIC as of possible use in various styles, and it is used by the completers which understand a `numeric' value in their relevant styles; the $WIDGET and $KEYS parameters are useful for deciding between different behaviours based on what the widget is called (as in _history_complete_word), or which keys are used to invoke it (as in _bash_completions).

Here are a few examples of using special parameters and compset.

One of the shortest standard completions is this, _precommand:

```
  #compdef - nohup nice eval time rusage noglob nocorrect exec

  shift words
  (( CURRENT-- ))

  _normal
```

It applies for all the standard commands which do nothing but evaluate their remaining arguments as a command, with some change of state, e.g. ignoring a certain signal (nohup) or altering the priority (nice). All the completion system does here is shift the first word off the end of the $words array, decrement the index of the current word into $words, and call _normal. This is the function called when completion occurs not in one of the special -context-s, in other words when an argument to an ordinary command is being completed. It will look at the new command word $words[1], which was previously the first argument to nohup or whatever, and start completion again based on that, or even complete that word itself as a command if necessary. The net effect is that the first word is ignored completely, as required.

Here's just an edited chunk of the file `_user_at_host`; as its name suggests, it completes words of the form `<user>@<host>`, and it's used anywhere the `user-hosts` style, described above, is appropriate:

```
  if [[ -prefix 1 *@ ]]; then
    local user=${PREFIX%%@*}

    compset -P 1 '*@'

    # complete the host for which we want the user
  else
    # no @, so complete the user
  fi
```

We test to see if there is already a `` `<user>@' `` part. If there is, we extract the user with an ordinary parameter substitution (so ordinary even other shells could do it). Then we strip off that from the bit to be completed with `compset`; we already know it matches the prefix, so we don't need to test the return value. Then we just do normal hostname completion on what remains --- except that the `user-hosts` style might be able to give us a clue as to which hosts have such a user. If the original test failed, then we simply complete what's there as a user.

Finally, here is essentially what the function `_most_recent_file` uses to extract the `$NUMERIC`th (default first) most recently modified file.

```
  local file
  file=($~PREFIX*$~SUFFIX(om[${NUMERIC:-1}]N))
  (( $#file )) && compadd -U -i "$IPREFIX" -I "$ISUFFIX" -f -Q - $file
```

Instead of doing it with mirrors, this uses globbing qualifiers to extract the required file; `om` specifies ordering by modification time, and the expression in square brackets selects the single match we're after. The `N` turns on `NULL_GLOB`, so `$file` is empty if there are no matches, and the parameter expansions with `` `$~' `` force patterns in `$PREFIX` and `$SUFFIX` to be available for expansion (a little extra feature I use, although ordinary completion would work without).

Most of the `compadd` command is bookkeeping to make sure the parts of the prefix and suffix we've already removed, if there are any, get passed on, but the reason for that deserves a mention, since normally this is handled automatically. The difference here is that `-U` usually replaces absolutely everything that was in the word before, so if you need to keep it you have to pass it back to `compadd`. For example, suppose you were in a context where you were completing after `` `file=... `` and you had told the completion system that everything up to `` `file=' `` was not to count and not to be shown as part of the completion. You would want to keep that when the word was put back on the command line. However, `` `-U' `` would delete that too. Hence the `` `-i "$IPREFIX"' `` to make sure it's retained. The same argument goes for the ignored suffix. However, there's currently no way of getting `_most_recent_file` to work on only a part of a string, so this explanation really only applies when you call it from another completion function, not directly from the command line.

6.9.6: Fancier completion: using the tags and styles mechanism

At this point, you should be in a position to construct, although maybe not in the best possible way, pretty much any completion list you want. Now I need to explain how you make sure it all fits in with the usual tags and styles system. You will need to pick appropriate tags for your completions. Although there is no real restriction, it's probably best to pick one of the standard tags, some of which are suitably general to cover just about anything: `files`, `options`, `values`, etc. There is a list in the completion system manual entry. Remember that the main use for tags is to choose what happens when more than one tag can be completed in the same place. Finding such things that can't be separated using the standard tag names is a good reason for inventing some new ones; you don't have to do anything special if the tag names are new, just make sure they're documented for anyone using the completion function.

How to call functions so that `It Just Works'

The simplest way of making your own completion function recognize tags is to use the `_description` function, which is usually called with three arguments: the name of the tag you're completing for, the name of a variable which will become an array containing arguments to pass to `compadd`, and the full description. Then you have to make sure that array gets passed down to `compadd`, or to any of the higher-level completion functions which will pass the arguments on to `compadd`. For example,

```
local expl
_description files expl 'my special files'
_files "$expl[@]"
```

This sets the files tag; `_description` sets `$expl` to pass on the description, and maybe other things such as a group name for the tag, in the appropriate format; we pass this down to `_files` which will use it for calling `compadd`. Generally, you will call `_description` for each time you call `compadd` or something that in turn calls `compadd`.

The `_description` function calls another function `_setup` to do much of the setting up of styles for the particular tag. Mostly, `_setup` is buried deeply enough that you don't need to worry about it yourself. Sometimes you can't do completion, and just want to print a message unconditionally to say so, irrespective of tags etc.; the function `_message` does this, taking the message as its sole argument.

There are two levels above that; these implement the tags mechanism in full. In `_description`, all that happens is that the user is informed what tag is coming up; there's no check what preferences the user has for tags (the first level), nor whether he wants tags to be split up using the labelling mechanism, e.g. picking out certain sorts of files using the labelled tag ``file:-myfiles'` to get the final tag ``file-myfiles'` (the second level).

To get this for simple cases you use the function `_wanted`. Unlike `_description`, it's an interface to the function that generates completion as well as a handler for tags --- that's so it can loop over the generated tags, checking the labels. The call above would now look like this:

```
_wanted files expl 'my special files' _files
```

Note that you now don't pass the `"$expl[@]"`, which hasn't even been set yet; `_wanted` will generate the string using the parameter name you say (here ``expl'`, as usual), and assume that the function generating the completions can use the result passed down to it. This is true of pretty much anything you are likely to want to use.

Note also the fact you need to pass ``_files'`, i.e. the function generating the completion. You can put pretty much any command line which generates completions here, down to a simple ``compadd'` expression. The reason it has to be here is the tag labelling business: `_wanted` could check whether the tag you specify, ``files'`, is wanted by the user and then return control to you, but it wouldn't be able to split up and loop over labelled tags set in this case for the `file-patterns` style and in other case by the `tag-order` style.

Unless you're really going into the bowels, `_wanted` is probably the lowest level you will want to use. I'd suggest you remember that one, and only go back and look at the other stuff if you need to do something more complicated.

If your function handles multiple tags, you need to loop over the different tags to find out which sort the tag order wants next. For this, you first need to tell the system which tags are coming up, using the `_tags` function with a list. Then you need to to test whether each tag in turn actually needs to be completed, and go on doing this until you run out of tags which need completions performing; the `_tags` function without arguments does this. Finally, you need to use `_requested`, which works a bit like `_wanted` but is made to fit inside the loop we are using. The end result looks like this:

```
local expl ret=1
_tags foo bar rod
```

```
  while _tags; do
    _requested foo expl "This is the description for tag foo" \
      compadd all foos completions && ret=0
    _requested bar expl "This is the description for tag bar" \
      compadd all bars completions && ret=0
    _requested rod expl "This is the description for tag rod" \
      compadd all rods completions && ret=0
    (( ret )) || return 0   # leave if matches were generated
  done
```

If you do include the completion function line as arguments, the loop over labels for the tag you specify is automatically handled as with `_wanted`. It may be a little confusing that both `_requested` and `_wanted` exist: the specific difference is that with `_requested` you call the `_tags` function yourself, whereas `_wanted` assumes the only valid tag is its argument and acts accordingly, and can be used only for simple, `one-shot' completions.

With `_requested`, unlike `_wanted`, you can separate out the arguments to the completion generator itself --- here `compadd` --- into a different statement, remembering the `"$expl[@]"` argument in that case. You can miss out the second and third arguments for `_requested` in this way. This time the loop which generates labels for tags is not performed, and you have to arrange it yourself, with the usual trade off of greater complexity for greater flexibility. To do this, there are two other functions: `_all_labels` and `_next_label`. The simpler case is with `_all_labels`, which just implements the loop over the labels using the same arguments as `_wanted`:

```
  _requested values &&
     _all_labels values expl 'values for my special things'  \
       compadd alpha bravo charlie delta echo foxtrot.
```

In case you haven't understood (and it's quite complicated, I'm afraid): the `_requested` looks at whether the tag you use has been asked for by the user. Having found out that it is, the `_all_labels` function calls the command `compadd` which actually adds the completions, but it does it in such a way as to take account of labelled tags --- you might have both a plain `` `values' `` tag and `` `values:-special' `` labelled tag, and `_all_labels` is needed to decide which is being used here. This last example is actually exactly what `_requested` does when given the `compadd` as argument, so it's only really useful when there is some code between the `_requested` and the `_all_labels`, for example to compute the strings to complete.

The most complicated case you are likely to come across is when inside the part of the tags loop which handles a particular tag (i.e. the `_requested` lines in the example above), you actually want to add more than one possible sort of completion. Then `_all_labels` is no longer enough, because completion needs to sort out the different things which are being added. This can also happen when there is only one valid tag, but that has multiple completions so that `_wanted` isn't any use. In this case you need to use `_next_label` inside a loop, which, as its names suggests, fixes up labels for the current tag and stops when it's found the right one. Here's a stripped down example which handles completion of messages from the `MH` mail handling system; you'll find it complete inside the function `_mh`.

```
  _tags sequences
  while _tags; do
    while _next_label sequences expl sequence; do
      compadd "$expl[@]" $(mark $foldnam 2>/dev/null |
                          awk -F: '{ print $1 }') && ret=0
      compadd "$expl[@]" reply next cur prev \
          first last all unseen && ret=0
      _files "$expl[@]" -W folddir -g '<->' && ret=0
    done
    (( ret )) || return 0
  done
```

Here's what's going on. The `_tags` call works just as it did in the first example I showed for that, deciding whether the tag in question, `sequences`, has been asked for; the tag name comes because MH allows you to define sets of messages called exactly `sequences'. The first `` `while' `` selects all values from `tag-order` where the `` `sequences' `` tag appears, with or without a label. The second `` `while' `` loop then sorts out any occurrences of labelled sequences to be presented to the user at the same time, i.e. given in the same element of the `tag-order` value array. The first `compadd` extracts from the folder (MH's name for a directory) identified by the function the names of any sequences you have defined; the second adds a lot of standard sequences --- although strictly speaking `unseen` isn't a standard sequence since you can name it yourself in `~/.mh_profile`. Finally, the third adds files in the folder itself whose names are just digits, which is how MH stores messages. The handling of `return` makes sure it stops as soon as you have matches for one particular element of `tag-order`; if you put it in the inner loop, you would just have the first of those sets that happened to be generated, while here, if you specify that all types of sequence should appear in the same completion list, they are all correctly collected.

Why, in that last example, is there no call to `_requested`, now I've gone to the trouble of explaining what that does? The answer is that there is only one tag; `_tags` can decide if we want it at all, and after that the tag is known, so we don't need `_requested` to find that information out for us. It's only needed if there is more than one type of match --- indeed, that's why we introduced it, so this is not actually a new complication, although you can be forgiven for thinking otherwise.

Here's an example of using that code for sequences. You might decide that you only want to see named sequences unless there aren't any, otherwise ordinary messages. You could do this by setting your styles as follows:

```
  zstyle ':completion:*' tag-order sequences:-name sequences:-num
  zstyle ':completion:*:sequences-name' ignored-patterns '(|,)<->'
  zstyle ':completion:*:sequences-num' ignored-patterns  '^<->'
```

which tries `sequences` under the labels `sequences-name` and `sequences-num`; which ignore completions which are all digits, and those which are not all digits, respectively. The slight twiddle in the pattern for `sequences-name` ignores messages marked for deletion as well, which have a comma stuck in front of the number (this is configurable, so your version of MH may be different).

All of `_description`, `_wanted`, `_requested`, `_all_labels` and `_next_label` take the options `-J` and `-V` to specify sorted or unsorted listings and menus, and the options `-1` and `-2` for removing consecutive duplicates or all duplicates. These are also options to `compadd`; the reason for handling them here is that they can be different for each tag, and the function called will set `expl` appropriately.

If your requirements are simple enough, you can replace that `_tags` loop above with a single function, `_alternative`. This takes a series of arguments each in the form `<tag>:<description>:<action>', with the first two in the form you now know, and the third an action. These are essentially the same as actions for the `_arguments` function, described below, except that the form `` `->state' ``, which says that the calling function will handle the action itself by using the value of the parameter `$state`, is not available. The most common forms of action here will be a call to another completion function, maybe with arguments (e.g. `` `_files -/' ``), or a simple list in parentheses (e.g. `` `(see saw margery daw)' ``). Here, for example, is how the `_cd` function handles the two cases of local directories (under the current directory) and directories reached via the `$cdpath` parameter:

```
  local tmpcdpath
  tmpcdpath=(${(@)cdpath:#.})
  _alternative \
      'local-directories:local directories:_path_files -/' \
      'path-directories:directories in cdpath:
  _path_files -W tmpcdpath -/'
```

The only tricky bit is that `$tmpcdpath`: it removes the `` `.' `` from `$cdpath`, if it's present, so that the current directory is always searched for with the tag `` `local-directories' ``, never with `` `path-directories' ``. Actually, you could argue that it should be treated as being in `` `path-directories' `` when it's present; but that confuses the issue over what `` `local-directories' `` really means, and it is useful to have the distinction.

It's now an easy exercise to replace the example function I gave for `_requested` by a call to `_alternative` with the arguments to `compadd` turned into a list in parentheses as the `<action>` part of the arguments to `_alternative`.

How to look up styles

If your completion function gets really sophisticated, you may want it to look up styles to decide what its behaviour should be. The same advice goes as for tags: only invent a new style if the old ones don't seem to cover the use you want to make, since by using contexts you can always restrict the scope of the style. However, by the same token don't try to squeeze too much meaning into one style, which will force the user to narrow the context --- it's always much easier to set a style for the general context `` `:completion:*' `` than to have to worry about all the circumstances where you need a particular value.

Retrieving values of styles is no harder than defining them, but you will need to know about the parameter `$curcontext`, which is what stores the middle part of the context, sans `` `:completion:' `` and sans tag. When you need to look something up, you pass this context to `zstyle` with `` `:completion:' `` stuck in front:

```
  zstyle -b ":completion:${curcontext}:tag" style-name parameter
```

If the tag is irrelevant, you can leave it empty, but you still need the final colon since there should always be six in total. In some cases where multiple tags apply it's useful to have a `:default` tag context as a fall back if none of the actual tags yield styles for that context; hence you should test the style first for the specific tag, then with the `default`.

Style lookups all have the form just shown; the result for looking up `style-name` in the given context will be saved in the `parameter` (which you should make local, obviously). In addition, `zstyle` returns a zero status if the lookup succeeded and non-zero if it failed. The `-t` lookup is different from the rest as it only returns a status for a boolean, i.e. returns status 0 if the value is `true`, `yes`, `1` or `on`, and doesn't require a parameter name. There is also a `-T`, which is identical except that it returns status 0 if the style doesn't exist, i.e. the style is taken to default to true.

The other lookup options return the style as a particular type in the parameter with exit status zero if the lookup succeeded, i.e. a value was found, and non-zero otherwise; `-b`, `-s`, and `-a` specify boolean (`parameter` is either `yes` or `no`), scalar (`parameter` is a scalar), and array (`parameter` is an array, which may still be a single word, of course), You can retrieve an associative array with `-a` as long as the parameter has already been declared as one.

There's also a convenience option for matching, `-m`; instead of a `parameter` this takes a `pattern` as the final argument, and returns status zero if and only if the `pattern` matches one of the values stored in the style for the given context.

Typical usages are thus:

```
  if zstyle -t ":completion:${curcontext}:" foo; then
    # do things in a fooish way
  else
    # do things in an unfooish way
  fi
```

or to use the value:

```
  local val
  if zstyle -s ":completion:${curcontext}:" foo val; then
    # use $val to establish how fooish to be
  else
    # be defaultly fooish
  fi
```

6.9.7: Getting the work done for you: handling arguments etc.

The last piece of unfinished completion business is to explain the higher level functions which can save you time writing completions for commands which behave in a standard way, with arguments and options. The good news is that all the higher functions here handle tags and labels internally, so you don't need to worry about `_tags`, `_wanted`, `_requested`, etc. There's one exception: the `state' mechanism to be described, where a function signals you that you're in a given state using the parameter `$state`, expects you to handle tag labels yourself --- pretty reasonable, as you have requested that the function return control to you to generate the completions. I've mentioned that here so that I don't have to gum up the description of the functions in this section by mentioning it again.

Handling ordinary arguments

The most useful function is `_arguments`. There are many examples of this in the completion functions for external commands, since so many external commands take the standard format of a command with options, some taking their own arguments, plus command arguments.

The basic usage is to call it with a series of arguments (which I'll call `specifications') like:

```
  <where I am>:<description>:<what action to take>
```

although there are a whole series of more complicated possibilities.

The initial ``<where I am>`' part tells the function whether the specification applies to an argument in a particular position, or to an option and possibly any arguments for that option. Let's start with ordinary arguments, since these are simpler. In this case ``<where I am>`' will be either a number, giving the number of the argument, or a ``*`', saying that this applies to all remaining arguments (or all arguments, if you haven't used any of the other form). You can simplify the first form, by just missing out the number; then the function will assume it applies to the first argument not yet specified. Hence the standard way of handling arguments is with a series of specifications just beginning ``:`' for arguments that need to be handled their own way, if any, then one beginning ``*:` for all remaining arguments, if any.

The message that follows is a description to be passed on down to `_description`. You don't specify the tags at this point; that comes with the action.

The action can have various forms, chosen to be easily distinguishable from one another.

1. A list of strings in parentheses, such as ``(red blue green)`'. These are the possible completions, passed straight down to `compadd`.
2. The same, but with double parentheses; the list in this case consists of the completion, a backslashed colon, and a description. So an extended version of the previous action is ``((red\:The\ colour\ red blue\:The\ colour\ blue))`' and so on. You can escape other colons inside the specifications in this way, too.
3. A completion function to call, with any arguments, such as ``_files -/`' to complete directories. Usually this does the business with `$expl` which should be familiar from the section on basic tag handling, however you can put an extra space in front of the action to have it called exactly as is, after word splitting.
4. A word preceded by ``->`' for example ``->state`'. This specifies that `_arguments` should return and allow the calling function to process the argument. To signal back to the calling function, the parameter `$state` will be set to what follows the ``->`'. It's up to the calling function to make `$state` a local parameter --- `_arguments` can't do that, since then it couldn't return a value.

   You should also make the parameters `$context` and `$line` local; the former is set to the new part to be added to `$curcontext`, which, as you can find out from `^Xh`, is `option-<option>-<arg>`, for example `option-file-1` for the first argument of the `option-file` option, or `argument-N`, for example `argument-2` for the second argument of the command.

   In simple cases, you will just test the parameter `$state` after `_arguments` has returned to see what to do: the return value is 300 to distinguish it from other returns where `_arguments` itself performed the completion.

5. A chunk of code to evaluate, given in braces, which removes the need for a special function or processing states. Obviously this is best used for the simplest cases.

These are the main possibilities, but I have not described every variation. As always, you should see the manual for all the detail.

Here's a concocted example for that `->state' action specifier, in case it's confusing you. It's for a command that takes arguments `alpha', `beta' and `gamma', and takes a single option `-type' which takes one argument, either `normal' or `unusual'.

```
local context state line
typeset -A opt_args

_arguments '-type[specify type]:type:->type' \
           '*:greek letter:->gklet' && return 0

case $state in
  (type)  compadd normal unusual && return 0
          ;;
  (gklet) compadd alpha beta gamma && return 0
          ;;
esac

return 1
```

In fact the possibilities here are so simple that you don't need to use $state; you can just use the form with the values in parentheses as the action passed to `_arguments'. Anyway, if you put this into a function `_foo', type `compdef _foo foo', and attempt completion for the fictitious command `foo', you will see _arguments in action.

I haven't shown the gory tag handling; as it's written, you'll see that no tag is ever defined for the compadd arguments shown. In this case you could just use _wanted. What you get for free with arguments, however, is the context: in the first case, you would have `:option-type-1' in the argument field (the second last, just before the tag), and in the second case `:argument-rest:'. Go back to where I originally described contexts if you've forgotten about these; I didn't tell you at the time, but it's the _argument function that is responsible for them. (However, you can supply a `-C' argument to _wanted to tell that a context.)

A note about the form: that `&& return 0' makes the completion function return if _arguments was satisfied that it found a completion on its own. It's useful in more complex cases. Remember that most completion functions return status zero if and only if matches were added; this function is written to follow that convention. I already showed this in the section on tags, but you might have skipped that.

Note all the things you had to make local: $context, $state, $line and the associative array $opt_args. The last named allows you to retrieve the values for a particular option; for example `$opt_args[-o]' contains any value already on the command line for the option -o. For options that take multiple arguments, these appear separated by colons, so if the line contains `-P prefix 3', $opt_args[-P] will contain `prefix:3'.

Handling options

Option handling is broadly similar, with the `<where I am>' part just giving the option name --- I already showed one example with `-type' above. In this case, the option will just be completed to itself, the first part of the specification, and the rest says how to complete its arguments. Since options can take any number of arguments, including zero, the :description:action pair can be repeated, or omitted entirely. Otherwise, it behaves similarly to the way described for ordinary command arguments, with all the same possible actions. So a simple option specification could be

```
_arguments '-turnmeon'
```

for an option with no arguments,

```
  _arguments '-file:input file:_files'
```

for an option with one argument, or

```
  _arguments '-iofiles:input file:_files:output file:_files'
```

for an option with two arguments, both files but with different descriptions.

The first part of the specification for an option can be more complicated, to reflect the fact that options can be used in all sorts of different ways. You can specify a description for the option itself --- as I tried to explain, the descriptions in the rest of the specification are instead for the arguments to the option. To specify an option description, just put that after the option, before any colons, in square brackets:

```
  _arguments '-on[turn me on, why not]'
```

Next, some options to a command are mutually exclusive. As `_arguments` has to read its way along the command line to parse it, it can record what options have already appeared, and can ensure that an option incompatible with one there already will not be completed. To do this, you need to include the excluded option in parentheses before the option itself:

```
  _arguments '(-off)-on[turn me on, why not]' \
             '(-on)-off[turn me off, please]'
```

This completes either of the options `` `-on' `` or `` `-off' ``, but if you've already given one, it won't complete the other on the same command line. If you need to give multiple excluded options, just list them separated by spaces, like `` `(-off -noton)' ``.

Some options can themselves be repeated; `_arguments` usually won't do that (in a sense, they are mutually exclusive with themselves), but you can allow it to happen by putting a `` `*' `` in front of the option specification:

```
  _arguments '*-o[specify extra options]:option string:->option'
```

allows you to complete any number of `` `-o <option>' `` sets using the `$state` mechanism. The `*` appears after any list of excluded options.

There are also ways of allowing different methods of option handling. If the option is followed by `-`, that means the value must be in the same word as the option, instead of in the next word; if that is allowed, but the argument could be in the next word instead, the option should be followed by a `` `+' ``. The latter behaviour is very common for commands which take single letter options. Some commands, particularly many recent GNU commands, allow you to have the argument in the next word or in the current word after an `` `=' `` sign; you get this by putting an `` `=' `` after the option name. For example,

```
  _arguments '-file=:input file:_files'
```

allows you to complete `` `-file `` *<filename>*' or `` `-file= ``*<filename>*'. With

```
   _arguments '-file=-:input file:_files'
```

only the second is possible, i.e. the argument must be after the `` `=' ``, not in its own word.

You can handle optional and repeated arguments to options, too. This illustrates some possibilities:

```
  _arguments '-option:first arg:->first::optional arg:->second'
```

The doubled colon indicates that the second argument is optional. In other words, at that point on the command line `_arguments` will either try to complete via the state `second`, or will try to start another specification entirely.

```
  _arguments '-option:first arg:->first:*:other args:->other'
```

Here, all arguments after the first --- everything else on the command line --- is taken as an argument to the option, to be completed using the state `other`.

```
  _arguments '-option:first arg:->first:*-:other args till -:->other'
```

This is similar, but less drastic: there is a pattern after the `*', here a `-', and when that is encountered, processing of arguments to `-option' stops. A command using this might be called as follows:

```
  cmdname -option <first> <other1> <other2> .... - <remainder>
```

where of course completion for `<remainder>` might be handled by other specifications.

There are yet more possible ways of handling options. I've assumed that option names can have multiple letters and hence must occur in separate words. You can specify single-letter options as well, of course, but many commands allow you to combine these into one word. To tell `_arguments` that's OK you should give it the option `-s`; it needs to come before any specifications, to avoid getting mixed up with them. After you specify this, a command argument beginning with a single `-' will be treated by `_arguments` as a list of single options, so `-lt' is treated the same as `-l -t'. However, options beginning with `--' are still treated as single options, so a `--prefix' on the command line is still handled as a single long option by `_arguments`.

One nice feature which can save a lot of trouble when using certain commands, notably those written by the GNU project and hence installed on most Linux-based systems, which take an option `--help' that prints out a list of all options. This is in a human-readable form, but `_arguments` is usually able to extract a list of available options which use the `--...' form, and even in many cases whether they take an argument, and if so what type that is. It knows because `<command> --help' often prints out a message like `--file=FILE' which would tell `_arguments` (1) that `--file' is a possible option (2) that it takes an argument because of the `=' (3) that that argument should be a file because of the message `FILE' at the end.

You specify that the command in question works in this way by using the (fairly memorable) option `--' to `_arguments'. You can then help it out with completion of option arguments by including a pattern to be matched in the help test after the `--'; the format is otherwise similar to a normal specification. For example `*=FILE*:file:_files' says that any option with `=FILE' in it has the description `file' and uses the standard `_files` function for completion, while `*=DIR*:directory:_files -/' does the same for directories. These two examples are so common that they are assumed by `_arguments --'.

So for example, here is the completion for `gdb`, the GNU debugger, which not surprisingly understands the GNU option format:

```
  _arguments -- '*=(CORE|SYM)FILE:core file:_files' \
                '*=EXECFILE:executable:_files -g \*\(\*\)' \
                '*=TTY:terminal device:compadd /dev/tty\*' && return 0
```

If you run `gdb --help', you'll see where these come from: `--core=COREFILE', `--exec=EXECFILE' and `--tty=TTY' are all listed as possible option/argument pairs. Doing it this way neatly allows the argument completions to work whatever the names of the options --- though of course it's possible for the rest of the pattern to change, too, and the commands, being written by lots of different people, are not necessarily completely consistent in the way their help text is presented.

6.9.8: More completion utility functions

This is now just a ragbag of other functions which might prove useful in your own completion functions, and which haven't been mentioned before, with some examples; once again, consult the manual for more detail. Note that many of these functions can take the most useful arguments to `compadd` and pass them on, even where I haven't explicitly said so.

`_call_function`

This is a simple front end to calling a function which may not be defined and hanging onto the return status of the function. One good use for this is to call a possibly non-existent function which might have been defined by the user, before doing some default stuff the user might want to skip. That would look like this:

```
  local ret  # returned status from called function, if it was called

  _call_function ret _hook_function arg1 arg2  &&  return ret

  # if we get here, _hook_function wasn't called,
  # so do the default stuff.
```

As you can work out, `_call_function` itself returns status zero if the function in the second argument got called, and in that case the first argument is the name of a parameter with the return status from the function itself. The whole point is that this is safe if `_hook_function` doesn't exist.

This function is too low level to know about the tags mechanism; use `_wanted` or similar to handle tags properly.

`_contexts`

This is another shorthand: the arguments it takes are a set of short contexts, in other words either names of commands or special contexts like `` `-math-' ``. The completion for each of these contexts is tried in turn; `_contexts` simply handles all the boring looking up of functions and testing the return values. The definition, if you want to look, is reassuringly simple. It only has one use at the moment: `_subscript`, which handles the `-subscript-` context we met early in the chapter, calls `` `_contexts -math-' `` to try mathematical completion, since ordinary array subscripts can contain mathematical expressions.

This is also too low level to handle tags. In zsh 4.1, it is made obsolete by a cleverer mechanism for handling different contexts which can be used, for example, for handling of arguments to redirections for particular commands, or keys in a particular associative array. I expect I'll describe that when 4.1 is finally released.

`_describe`

Don't confuse this with `_description` which was explained above and is the basic function for adding a description to a set of completions of a certain type. I mentioned in the description of the `verbose` style that this function was responsible for showing, or not showing, the descriptions for a whole lot of options at once. It allows you to do that with several different sets of completions that may require different options to `compadd`. The general form looks something like this:

```
  _describe "description of set 1" descs1 compls1 \
               <compadd-opts-1> -- \
            "description of set 2" ...
```

where you can have any number of sets separated by the `` `--' ``. The `descs1` and `compls1` are arrays of the same length, giving a list of descriptions and a list of completions, respectively. Alternatively, you need only give one array name and each element of that will contain a completion and a description separated by the now-traditional colon. The `` `<compadd-opts-1>' `` are a set of any old options recognised by `compadd`, such as `-q`, or `-S=/`, or what have you. I won't give an example for this, since to find something requiring it would almost need me to rewrite the completion system from scratch.

`_combination`

This is the function at the heart of the completions such as `users-hosts` described above, where combinations of elements need to be completed at the same time. It's easiest to describe with an example; let's pick the `users-hosts` example, and I'll assume you remember how that works from the user's point of view, including the format of the `users-hosts` style itself. The completion for the username part is performed as:

```
  _combination my-accounts users-hosts users
```

where `my-accounts` is the tag to be used for the completion, then comes the style, and then the part of the style to be extracted.

Now suppose we come back into the completion function again to complete the host later on the command line, so that the username is already there. We can find that by searching the command line; suppose we store what we find in `$userarg`. Then we can complete the hostname as follows:

```
  _combination my-accounts users-hosts users=$userarg hosts
```

and the magic part, the fact that we can limit the hostnames to be completed to only those with a user `$userarg`, is handled by `_combination`. This extends to `hosts-ports-users` and any larger combined set in the obvious way: the first field not to contain an `=' is the one being completed. You don't need to supply other fields if they are not known; in other words, the field to be completed doesn't need to be the first one in sequence not known, it can be any, just as long as it matches part of the style given in the second argument, so you could have omitted the `` `users=$userarg' `` in the last example if you couldn't extract the right username.

There are various bells and whistles: after the field to be completed you can add any options to be passed down to `compadd`; you can give `_combination` itself the option `` `-s <sep>' `` to specify a character other than colon to separate the parts of the style values; if the style lookup fails, but there is a corresponding function, which would be called `` `_users' `` or `` `_hosts' `` in this example, it is called to generate the matches, and gets the options at the end which are otherwise destined for `compadd`.

As you can see, this function is at a high enough level to handle the tags mechanism itself.

`_multi_parts`

This takes two arguments, a separator and a list of matches. The list of matches is normal, except that each element is likely to contain the separator. In the most obvious usage, the separator is `` `/' `` and the list of matches is a lot of files with path components. Here's another reasonable usage:

```
  local groups expl
  groups=($(awk -F: '{ print $1 }' ~/.newsrc))
  _wanted groups expl 'newsgroup' _multi_parts "$expl[@]" . groups
```

The generated array contains names of Usenet newsgroups, i.e. names with components separated by a `` `.' ``, and `_multi_parts` allows you to complete these piece by piece instead of in one go. This is a good deal better for use with menu completion, and the list which appears is smaller too. The `_wanted` part handles the tags mechanism, which `_multi_parts` doesn't.

`_sep_parts`

This also completes a word piece by piece, but unlike `_multi_parts` the trial completions are also only supplied for each piece. The arguments are alternating arrays and separators; arrays are in the usual form, in other words either the name of an array parameter, or a literal array in parentheses, quoted to protect it from immediate shell expansion. The separators are simply strings. For example

```
  local expl
  array1=(apple banana cucumber)
  _wanted breakfast expl 'breakfast' \
    _sep_parts array1 + '(bread toast croissant)' @ '(bowl plate saucer)';
```

completes strings like `` `apple+toast@plate' ``, piece by piece. This is currently not used by the distributed completion code.

`_values`

This works a little like `_arguments`, but is designed for completing the values of a single argument in a form like `` `key=val,flag,key=other' ``, in which you can specify the list separator, here `` `,' `` by using the option `-s`, e.g. `` `-s ``

,'. The first argument to `_values` is the overall description of the set of arguments. The other arguments are very much like those to `_arguments` except that, as you would expect from the form given, no pluses or minus signs are involved and each value can only have one argument, which must follow an `='. Virtually everything else is identical, with the exception that the associative array where the arguments are stored for each value is called `$val_args`.

I won't bother giving the instructions for `_arguments` again; instead, here is an example based on the values used by the `-o` option to the `mount` command:

```
local context state line
typeset -A val_args

_values -s , 'file system options' \
  '(rw)ro[mount file system read-only]' \
  '(ro)rw[mount file system read-write]' \
  'uid[set owner of root]:user ID:' \
  'gid[set group of root]:group ID:' \
  'bs[specify block size]:block size:(512 1024 2048 4192)'
```

I've just picked out a few of the umpteen possibilities for illustration; see the function `_mount` if you want more. Remember that the `` `(rw)' `` before the `` `ro' `` means that the options are mutually exclusive, and the one in parentheses won't be offered if the other appears on the command line; the strings in square brackets are descriptions of the particular options; and if there is a colon after the name of the value, the value takes an argument whose own description comes next. The second colon is followed by possible completions for that argument, using the usual convention for actions in `_arguments`; as you'll see from the `local` statement, the `$state` mechanism can be used here. Only the `` `bs' `` argument here is given possible completions; for `uid` and `gid` you'll have to type in the number without completion; `ro` and `rw` don't take arguments.

Hence a typical(?) list to be completed by this would be `` `rw,uid=123,bs=2048' ``.

Remember also that you can use a `` `*' `` before the option name to say that it can appear more than once in the value list. The `_values` function handles the context and tags in a similar way to `_arguments`.

`_regex_arguments`

This function is for use when the behaviour of a set of command arguments is so complicated that even `_arguments` can't help. It allows you to describe the arguments as a regular expression (i.e. a pattern). I won't explain it because I haven't yet figured out how it works. If you think you need to use it, look at the manual entry and then at the `_apt` function which is currently its main application.

6.10: Finally

Completion is big and complex: this means that there are probably lots of bugs around, and things that I haven't described simply enough or which may be implemented in too complicated a way. Please send the `zsh-workers` mailing list any reports or constructive criticism on the subject.

Last of all, remember that the new completion system is ideally just supposed to work without you needing to worry exactly how. That's a bold hope, but at least much of the time you should be able to get away with using just the tab key and ordinary characters.

- Table of Contents
- Previous Chapter
- Next Chapter
